Turing Machines

Reading

- Malik Magdon-Ismail. Discrete Mathematics and Computing.
  - Chapter 26

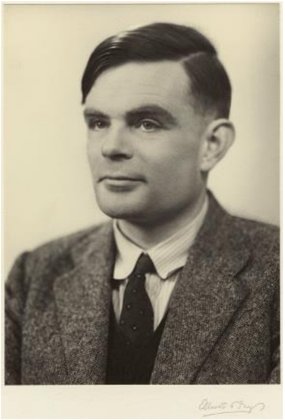
Overview

- Solving a non context free language: $w\#w$
- Transducer Turing Machines.
- Infinite Loops
- Encodings of Turing Machines

Turing’s 1936 Miracle

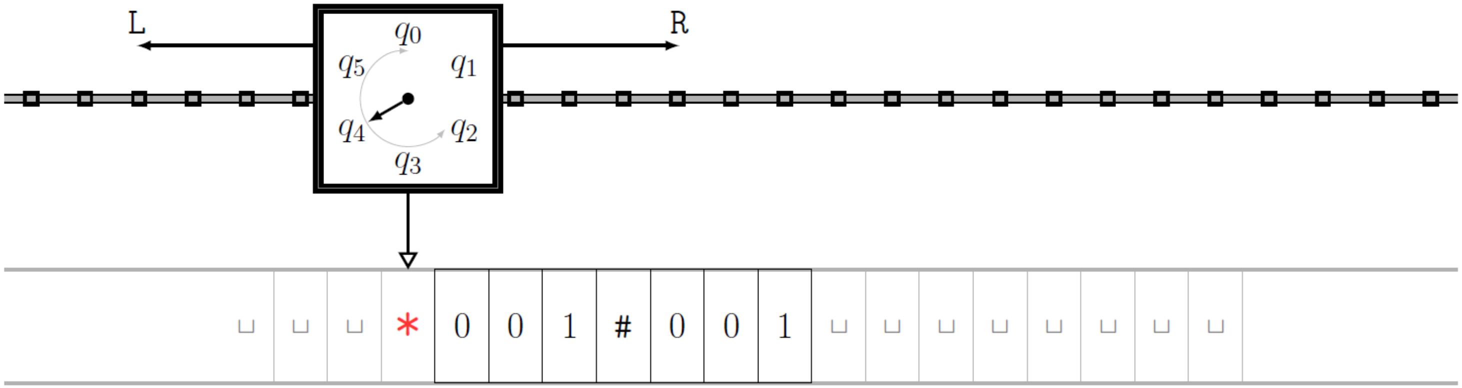
Rensselaer

- “On Computable Numbers with an Application to the Entscheidungsproblem”
  - Entscheidungsproblem was a question posed by Hilbert and Ackermann asking whether every logical proposition could be deduced from anxioms
- A classic which epitomizes the beauty of pure thought, where Alan Turing
  - Invented a notion of what it means for a number to be computable.
  - Invented the computer.
  - Invented and used subroutines.
  - Invented the *programmable* computer.
  - Gave a negative answer to Hilbert’s Entscheidungsproblem.



- All this before the world even saw its first computer. Wow!
- (Oh, and by the way, he helped the Alliance win WWII by decrypting the Enigma machine used for communication by the nazis.)

Turing's Machine

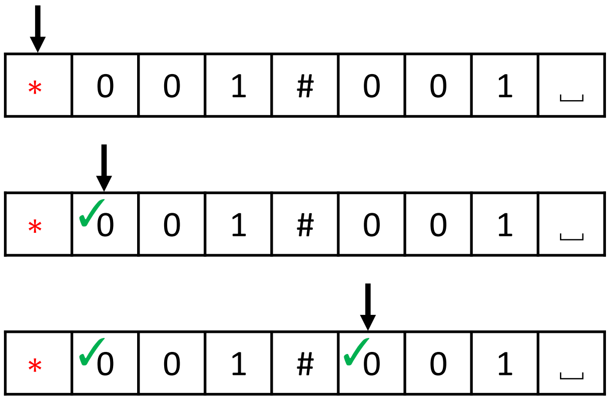


- States.
- Can move L/R (or stay put) giving random access to an infinite read-write tape.
- Input written on the tape to start.
- Instructions specify what to do based on state and what is on the tape.
- Beacon symbol $*$ (start of the tape).
- Let's see the capabilities of this machine on the non context free problem

$$\mathcal{L} = \{w\#w\}$$

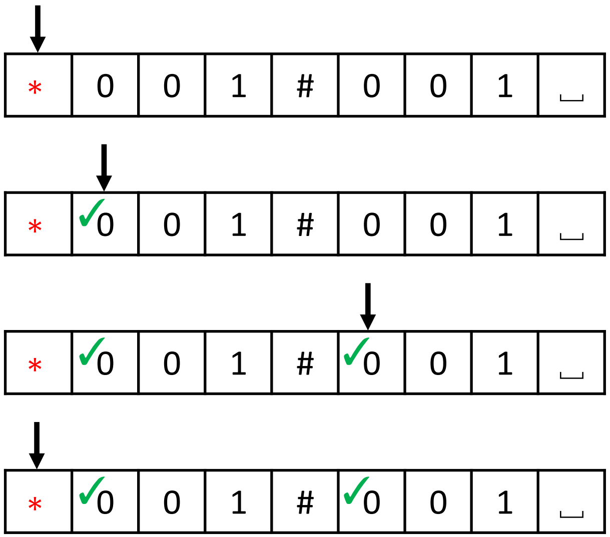
Solving *w#w*

1. Check for one "#", otherwise REJECT (a DFA can do this)
2. Return to "*".
3. Move right to first non-marked bit *before* "#".
   - Mark the location and *remember* the bit.
   - (If you reach "#" before any non-marked bit, goto step 5.)
4. Move right to first non-marked bit *after* "#".
   - If you reach "␣" before any non-marked bit, REJECT
   - If the bit does not match the bit from step 3, REJECT
   - Otherwise (bit matches), mark the location. goto step 2.



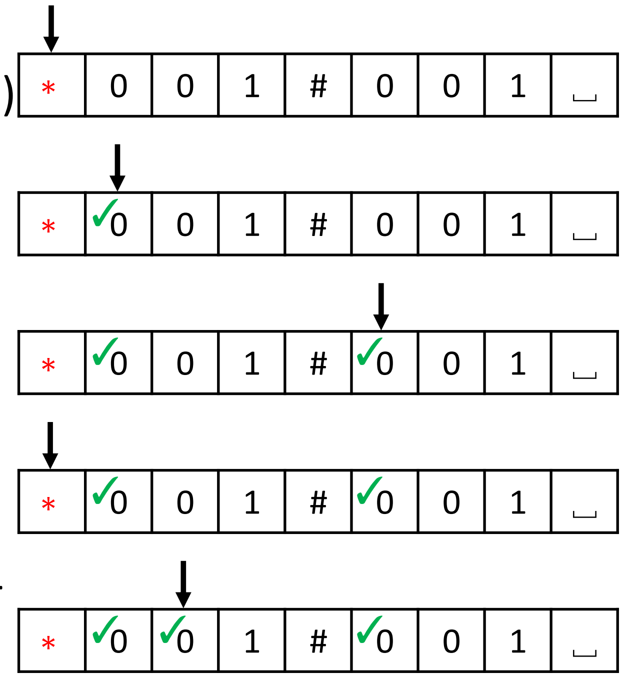
Solving *w#w*

1. Check for one "#", otherwise REJECT (a DFA can do this)
2. **Return to "*".**
3. Move right to first non-marked bit *before* "#".
   - Mark the location and *remember* the bit.
   - (If you reach "#" before any non-marked bit, goto step 5.)
4. Move right to first non-marked bit *after* "#".
   - If you reach "␣" before any non-marked bit, REJECT
   - If the bit does not match the bit from step 3, REJECT
   - Otherwise (bit matches), mark the location. goto step 2.



Solving *w#w*

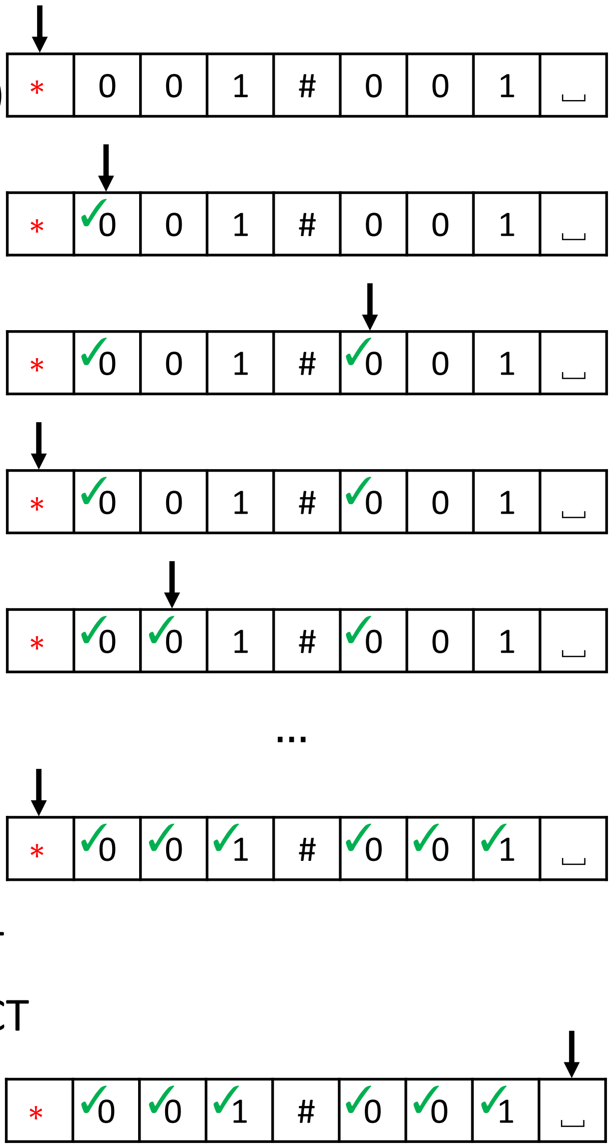
1. Check for one "#", otherwise REJECT (a DFA can do this)
2. Return to "∗".
3. **Move right to first non-marked bit *before* "#".**
   - **Mark the location and *remember* the bit.**
   - **(If you reach "#" before any non-marked bit, goto step 5.)**
4. Move right to first non-marked bit *after* "#".
   - If you reach "␣" before any non-marked bit, REJECT
   - If the bit does not match the bit from step 3, REJECT
   - Otherwise (bit matches), mark the location. goto step 2.



Solving *w*#*w*

1. Check for one "#", otherwise REJECT (a DFA can do this)
2. Return to "∗".
3. Move right to first non-marked bit *before* "#".
   - Mark the location and *remember* the bit.
   - (If you reach "#" before any non-marked bit, goto step 5.)
4. Move right to first non-marked bit *after* "#".
   - If you reach "␣" before any non-marked bit, REJECT
   - If the bit does not match the bit from step 3, REJECT
   - Otherwise (bit matches), mark the location. goto step 2.
5. Move right to first non-marked bit *after* "#".
   - If you reach "␣" before any non-marked bit, ACCEPT
   - If you find a bit (string on the right is too long), REJECT

- YES

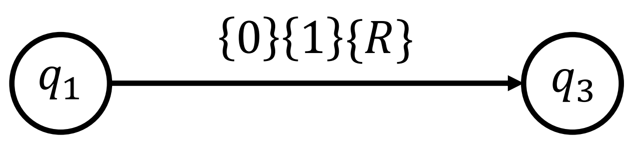


Turing Machine Instructions

- Similar to a DFA but with read/write capability
- Let's look at a DFA-like instruction: $q_1 0 q_3$
  - If in state q_1 and read 0, transition to q_3
- A Turing Machine can also write (and move its position) in addition to reading and transitioning

$$\{q_1\}\{0\} \rightarrow \{q_3\}\{1\}\{R\}$$

current state read next state write move

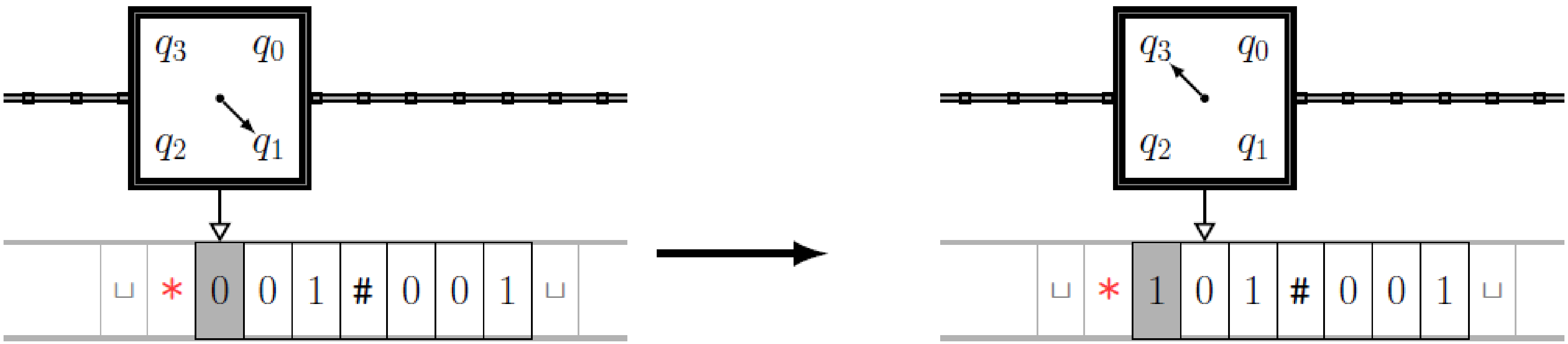


Turing Machine Instructions

- Similar to a DFA but with read/write capability
- Let's look at a DFA-like instruction: $q_1 0 q_3$
  - If in state q_1 and read 0, transition to q_3
- A Turing Machine can also write (and move its position) in addition to reading and transitioning

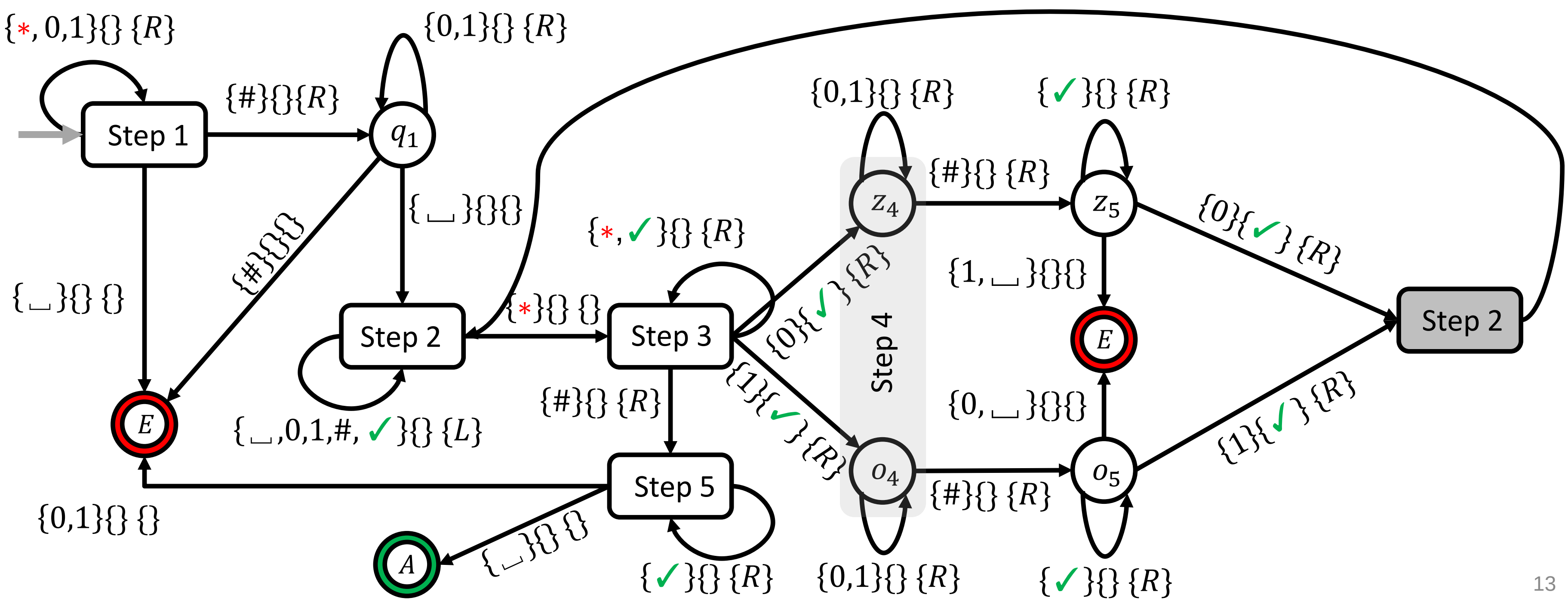
$$\{q_1\}\{0\} \rightarrow \{q_3\}\{1\}\{R\}$$

current state read next state write move



Building the Turing Machine that Solves $w\#w$

1. Check for one "#", otherwise reject (a DFA can do this)
2. Return to "∗".
3. Move right to first non-marked bit *before* "#".
   Mark and *remember* the bit. If you reach "#" before any non-marked bit, goto step 5.
4. Move right to first non-marked bit *after* "#".
   If you reach "␣" or bit doesn't match, REJECT. Otherwise, mark the location. goto step 2.
5. Move right to first non-marked bit *after* "#".
   If you reach "␣" ACCEPT. If you come to a non-marked bit, REJECT



Turing Machine for Multiplication

Rensselaer

- Consider a language that encodes multiplication

$$\mathcal{L}_{mult} = \left\{0^{\bullet i}\#1^{\bullet j}\#0^{\bullet k} \middle| i, j > 0 \; and \; k = i \times j\right\}$$

- Multiplication is repeated addition.
- Pair each left-0 with a block of right-0s equal to the number of 1s

	↓														
␣	*	0	0	#	1	1	1	#	0	0	0	0	0	0	␣

␣	*	✓0	0	#	1	1	1	#	0	0	0	0	0	0	␣

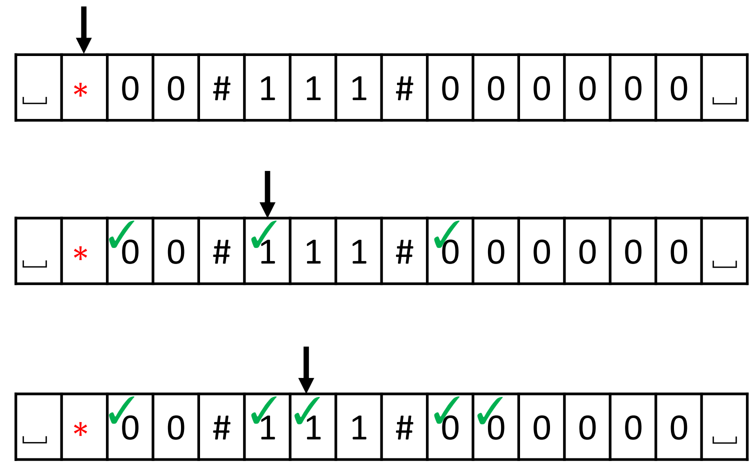
1. Verify the input format is $0^{\bullet i}\#1^{\bullet j}\#0^{\bullet k}$
   - (A DFA can solve this).
2. Return to $*$
3. Move right to mark first unmarked left-0, then right to "#".
   - If no unmarked left-0's (you reach "#"), goto step 6.

6. Move right to verify there are no unmarked right-zeros.
   - If you come to unmarked right-zero, REJECT; if come to " ␣ " ACCEPT

Turing Machine for Multiplication

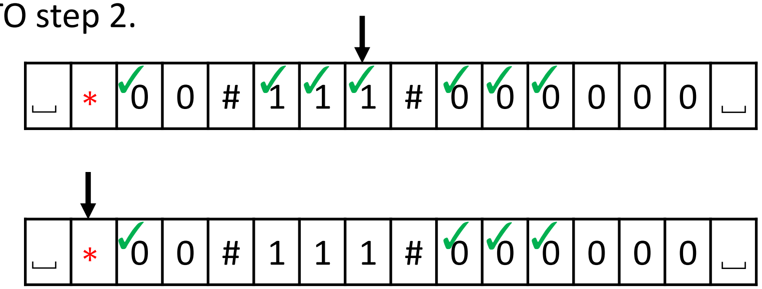
- Consider a language that encodes multiplication

$$\mathcal{L}_{mult} = \{0^{\bullet i} \# 1^{\bullet j} \# 0^{\bullet k} \mid i, j > 0 \ and \ k = i \times j\}$$



- Multiplication is repeated addition.
- Pair each left-0 with a block of right-0s equal to the number of 1s

1. Verify the input format is $0^{\bullet i} \# 1^{\bullet j} \# 0^{\bullet k}$
   – (A DFA can solve this).
2. Return to $*$
3. Move right to mark first unmarked left-0, then right to "#".
   – If no unmarked left-0's (you reach "#"), goto step 6.
4. Move right and mark first unmarked 1.
   – If all 1's marked (reach "#") move left, unmarking 1's. GOTO step 2.
5. Move right to find an unmarked right-0.
   – If no unmarked right-0's (come to "␣"), REJECT
   – Otherwise, mark the 0, move left to first marked 1. GOTO step 4.
6. Move right to verify there are no unmarked right-zeros.
   – If you come to unmarked right-zero, REJECT; if come to "␣" ACCEPT



Transducer Turing Machine that Multiplies

- Suppose that instead of checking if the multiplication is correct, we want a Turing Machine that performs the multiplication

␣	*	0	0	#	1	1	1	#	␣	␣	␣	␣	␣	␣	␣

→

␣	*	0	0	#	1	1	1	#	0	0	0	0	0	0	␣

- Algorithm is basically the same
- Instead of marking right-0s, write
- Turing Machines that modify the input are called transducers

␣	*	✓0	0	#	✓1	1	1	#	0	␣	␣	␣	␣	␣	␣

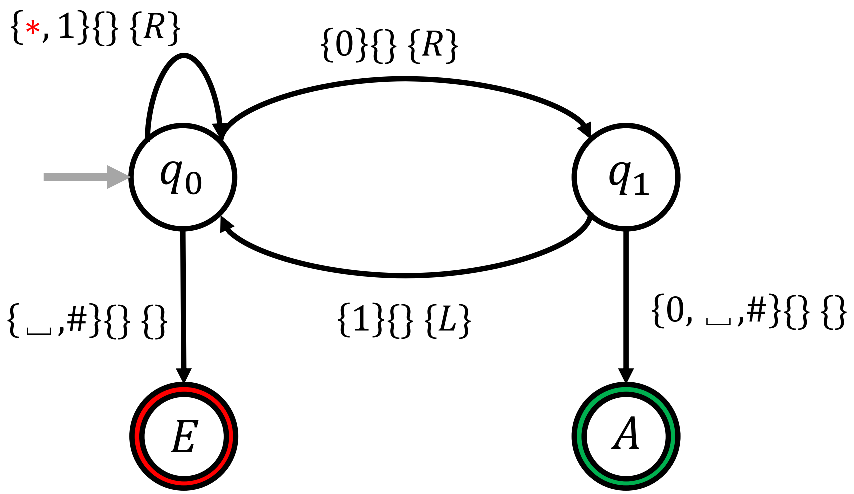
␣	*	✓0	0	#	✓1	✓1	1	#	0	0	␣	␣	␣	␣	␣

␣	*	✓0	0	#	✓1	✓1	✓1	#	0	0	0	␣	␣	␣	␣

␣	*	✓0	0	#	1	1	1	#	0	0	0	␣	␣	␣	␣

Infinite Loops

- Consider this Turing Machine



$$M(w) = \begin{cases} Halts\ in\ accept\ state \rightarrow ACCEPT \\ Halts\ in\ reject\ state \rightarrow REJECT \\ Loops\ forever \rightarrow\ ? \end{cases}$$

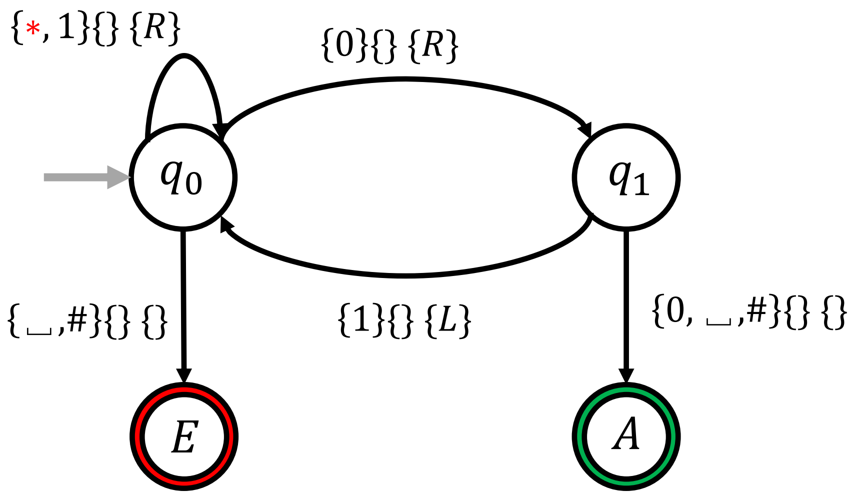
- What happens if the input is 01?
- Turing Machine M is a *recognizer* for language $\mathcal{L}(M)$:

$$w \in \mathcal{L}(M) \leftrightarrow M(w) = \text{halt with a YES}$$

$$w \notin \mathcal{L}(M) \leftrightarrow M(w) = \text{halt with a NO } \textit{or} \text{ loop forever}$$

Infinite Loops, cont'd

- Consider this Turing Machine



$$M(w) = \begin{cases} Halts\ in\ accept\ state \rightarrow ACCEPT \\ Halts\ in\ reject\ state \rightarrow REJECT \\ Loops\ forever \rightarrow\ ? \end{cases}$$

- What happens if the input is 01?
- Turing Machine M (not the one above) is a *decider* for language $\mathcal{L}(M)$:

$$w \in \mathcal{L}(M) \leftrightarrow M(w) = \text{halt with a YES}$$

$$w \notin \mathcal{L}(M) \leftrightarrow M(w) = \text{halt with a NO}$$

- **Practical *algorithms* must halt! Practical algorithms correspond to deciders.**

Mathematical Description of a Turing Machine

Rensselaer

- **States Q.** The first state is the start state, the halting states are A,E
- **Symbols Σ.** By default these are $\{*, 0, 1, \sqcup, \#\}$.
- **Machine-level transition instructions.** Each instruction has the form

  `{state}{read-symbol}{next-state}{written-symbol}{move}`

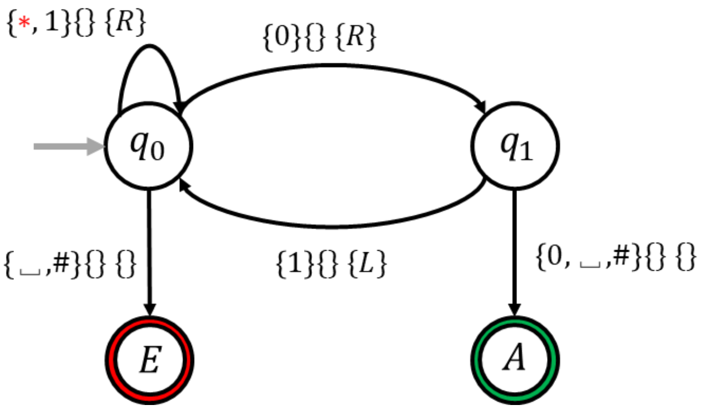
  – The instructions map each `(state,symbol)` pair to a `(state,symbol,move)` triple and thus form a *transition function*

$$\delta: Q \times \Sigma \rightarrow Q \times \Sigma \times \{L, R, S\}$$

Encoding a Turing Machine as a Bit-String

- **States**. $\{q_0, q_1, A, E\}$
- **Symbols**. $\{*, 0, 1, ␣, \#\}$
- **Machine-level transition instructions**.

$\{q_0\}\{*\}\{q_0\}\{*\}\{R\}$
$\{q_0\}\{1\}\{q_0\}\{1\}\{R\}$
$\{q_0\}\{0\}\{q_1\}\{0\}\{R\}$
$\{q_0\}\{\#\}\{E\}\{\#\}\{S\}$
$\{q_0\}\{␣\}\{E\}\{␣\}\{S\}$
$\{q_1\}\{1\}\{q_0\}\{1\}\{L\}$
$\{q_1\}\{0\}\{A\}\{0\}\{S\}$
$\{q_1\}\{\#\}\{A\}\{\#\}\{S\}$
$\{q_1\}\{␣\}\{A\}\{␣\}\{S\}$
$\{q_1\}\{*\}\{q_1\}\{*\}\{R\}$



- The description of a Turing Machine is a *finite* binary string.
- Turing machines are countable and can be listed: $\{M_1, M_2, \ldots\}$
- The problems solvable by an algorithm are countable: $\{\mathcal{L}(M_1), \mathcal{L}(M_2), \ldots\}$

Not all languages can be decided

- The description of a Turing Machine is a *finite* binary string.
- Turing machines are countable and can be listed: $\{M_1, M_2, \dots\}$
- The problems solvable by an algorithm are countable: $\{\mathcal{L}(M_1), \mathcal{L}(M_2), \dots\}$

- Can we list all languages?
  - Each language $\mathcal{L}(M)$ can be mapped to an infinite binary string

$\mathcal{B}$	ε	0	1	00	01	10	11	000	…
$\mathcal{L}(M)$	0	0	1	1	1	0	1	1	…

  - The set of all languages is uncountable!
  - **Some languages cannot be decided by a Turing Machine!**