Proof of the Pumping Lemma, PDA $\Rightarrow$ CFG, and Turing Machines

204213 Theory of Computation

Jittat Fakcharoenphol

Kasetsart University

December 17, 2008

Outline

1. Proof of the Pumping Lemma
2. PDAs $\Rightarrow$ CFGs
3. Turing Machines

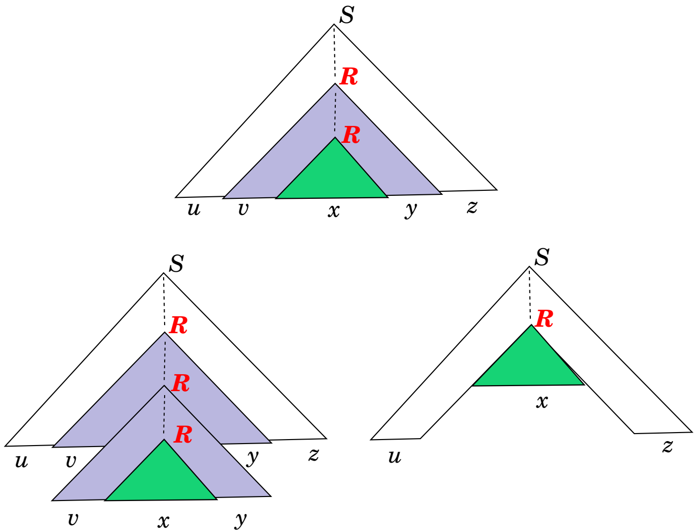
Pumping lemma for CFL

Theorem 1 (pumping lemma for CFL)

If A is a context-free language, then there is a pumping length p such that for any string $s \in A$ of length at least p, s can be divided into 5 pieces $s = uvxyz$ satisfying the following conditions

1. *for each $i \geq 0$, $uv^ixy^iz \in A$,*
2. *$|vy| > 0$, and*
3. *$|vxy| \leq p$.*

Parse tree for s



Proof Idea

- Recall our proof for the pumping lemma for regular languages.

Proof Idea

- Recall our proof for the pumping lemma for regular languages.
- How do we know that the DFA would visits the same state twice?

Proof Idea

- Recall our proof for the pumping lemma for regular languages.
- How do we know that the DFA would visits the same state twice?
- This time, we want the same variable to appear twice on some path in the parse tree.

Proof Idea

- Recall our proof for the pumping lemma for regular languages.
- How do we know that the DFA would visits the same state twice?
- This time, we want the same variable to appear twice on some path in the parse tree.
- Any idea? (Any hint?)

Proof Idea

- Recall our proof for the pumping lemma for regular languages.
- How do we know that the DFA would visits the same state twice?
- This time, we want the same variable to appear twice on some path in the parse tree.
- Any idea? (Any hint?) Yes, the Pigeon-Hole Principle.

Proof Idea

- Recall our proof for the pumping lemma for regular languages.
- How do we know that the DFA would visits the same state twice?
- This time, we want the same variable to appear twice on some path in the parse tree.
- Any idea? (Any hint?) Yes, the Pigeon-Hole Principle.
    - What?

Proof Idea

- Recall our proof for the pumping lemma for regular languages.
- How do we know that the DFA would visits the same state twice?
- This time, we want the same variable to appear twice on some path in the parse tree.
- Any idea? (Any hint?) Yes, the Pigeon-Hole Principle.
  - What?
  - Since we have $|V|$ variables, if, on the parse tree, the path from the start variable to some terminal is long enough (how long?)

Proof Idea

- Recall our proof for the pumping lemma for regular languages.
- How do we know that the DFA would visits the same state twice?
- This time, we want the same variable to appear twice on some path in the parse tree.
- Any idea? (Any hint?) Yes, the Pigeon-Hole Principle.
  - What?
  - Since we have $|V|$ variables, if, on the parse tree, the path from the start variable to some terminal is long enough (how long?) we should see the same variable twice.

Proof Idea

- Recall our proof for the pumping lemma for regular languages.
- How do we know that the DFA would visits the same state twice?
- This time, we want the same variable to appear twice on some path in the parse tree.
- Any idea? (Any hint?) Yes, the Pigeon-Hole Principle.
    - What?
    - Since we have $|V|$ variables, if, on the parse tree, the path from the start variable to some terminal is long enough (how long?) we should see the same variable twice.
    - How can we make sure that the parse tree is very tall?

Tall parse tree: example

G_1

$$
\begin{aligned}
S &\rightarrow AB \\
A &\rightarrow 1A0|0A1|\varepsilon \\
B &\rightarrow BB|0|1
\end{aligned}
$$

Tall parse tree: example

G_1

$$
\begin{array}{rcl}
S & \rightarrow & AB \\
A & \rightarrow & 1A0|0A1|\varepsilon \\
B & \rightarrow & BB|0|1
\end{array}
$$

- What is the longest string whose longest path from S to any terminal is ≤ 4?
- Any bound on the length of the string generated by G_2 that guarantees that the height of its parse tree is at least 5?

Tall parse tree: general case

- Let G be a CFG for CFL A.
- Let b be the maximum number of symbols generated from one varible by one of the substitution rules.

Tall parse tree: general case

- Let G be a CFG for CFL A.
- Let b be the maximum number of symbols generated from one varible by one of the substitution rules. (E.g., in G_1, b is 3.)

Tall parse tree: general case

- Let G be a CFG for CFL A.
- Let b be the maximum number of symbols generated from one varible by one of the substitution rules. (E.g., in G_1, b is 3.)
- Thus, any node on a parse tree of grammar G can't have more than b children.

Tall parse tree: general case

- Let G be a CFG for CFL A.
- Let b be the maximum number of symbols generated from one varible by one of the substitution rules. (E.g., in G_1, b is 3.)
- Thus, any node on a parse tree of grammar G can't have more than b children.
- Therefore, a tree of height 1 can't have more than b leaves.

Tall parse tree: general case

- Let G be a CFG for CFL A.
- Let b be the maximum number of symbols generated from one varible by one of the substitution rules. (E.g., in G_1, b is 3.)
- Thus, any node on a parse tree of grammar G can't have more than b children.
- Therefore, a tree of height 1 can't have more than b leaves. A tree of height 2 can't have more than

Tall parse tree: general case

- Let G be a CFG for CFL A.
- Let b be the maximum number of symbols generated from one varible by one of the substitution rules. (E.g., in G_1, b is 3.)
- Thus, any node on a parse tree of grammar G can't have more than b children.
- Therefore, a tree of height 1 can't have more than b leaves. A tree of height 2 can't have more than b^2 leaves.

Tall parse tree: general case

- Let G be a CFG for CFL A.
- Let b be the maximum number of symbols generated from one varible by one of the substitution rules. (E.g., in G_1, b is 3.)
- Thus, any node on a parse tree of grammar G can't have more than b children.
- Therefore, a tree of height 1 can't have more than b leaves. A tree of height 2 can't have more than b^2 leaves. In general a tree of height h can't have more than

Tall parse tree: general case

- Let G be a CFG for CFL A.
- Let b be the maximum number of symbols generated from one varible by one of the substitution rules. (E.g., in G_1, b is 3.)
- Thus, any node on a parse tree of grammar G can't have more than b children.
- Therefore, a tree of height 1 can't have more than b leaves. A tree of height 2 can't have more than b^2 leaves. In general a tree of height h can't have more than b^h leaves.

Tall parse tree: general case

- Let G be a CFG for CFL A.
- Let b be the maximum number of symbols generated from one varible by one of the substitution rules. (E.g., in G_1, b is 3.)
- Thus, any node on a parse tree of grammar G can't have more than b children.
- Therefore, a tree of height 1 can't have more than b leaves. A tree of height 2 can't have more than b^2 leaves. In general a tree of height h can't have more than b^h leaves.
- The length of a string is the number of leaves on a parse tree.

Tall parse tree: general case

- Let G be a CFG for CFL A.
- Let b be the maximum number of symbols generated from one varible by one of the substitution rules. (E.g., in G_1, b is 3.)
- Thus, any node on a parse tree of grammar G can't have more than b children.
- Therefore, a tree of height 1 can't have more than b leaves. A tree of height 2 can't have more than b^2 leaves. In general a tree of height h can't have more than b^h leaves.
- The length of a string is the number of leaves on a parse tree.
- Therefore a parse tree of a string of length $b^h + 1$ must be at least $h + 1$ high.

The proof (1)

- Let V be the set of variables of G. Let the pumping length p be $b^{|V|} + 1$.

The proof (1)

- Let V be the set of variables of G. Let the pumping length p be $b^{|V|}+1$.
- Consider any string $s \in A$ whose length is at least p.

The proof (1)

- Let V be the set of variables of G. Let the pumping length p be $b^{|V|} + 1$.
- Consider any string $s \in A$ whose length is at least p. From prev. slide, we know that any parse tree τ of s must be at least $|V| + 1$ high.

The proof (1)

- Let V be the set of variables of G. Let the pumping length p be $b^{|V|} + 1$.
- Consider any string $s \in A$ whose length is at least p. From prev. slide, we know that any parse tree τ of s must be at least $|V| + 1$ high.
- Since τ is at least $|V| + 1$ high, it must contain a path from root to some leaf of length $|V| + 1$.

The proof (1)

- Let V be the set of variables of G. Let the pumping length p be $b^{|V|} + 1$.
- Consider any string $s \in A$ whose length is at least p. From prev. slide, we know that any parse tree τ of s must be at least $|V| + 1$ high.
- Since τ is at least $|V| + 1$ high, it must contain a path from root to some leaf of length $|V| + 1$.
- There are $|V| + 2|$ nodes in this path; one is a terminal and $|V| + 1$ are variables.

The proof (1)

- Let V be the set of variables of G. Let the pumping length p be $b^{|V|}+1$.
- Consider any string $s \in A$ whose length is at least p. From prev. slide, we know that any parse tree τ of s must be at least $|V|+1$ high.
- Since τ is at least $|V|+1$ high, it must contain a path from root to some leaf of length $|V|+1$.
- There are $|V|+2|$ nodes in this path; one is a terminal and $|V|+1$ are variables.
- From PHP, since we have $|V|$ variables, some variable R appears more than once on this path.

The proof (2)

- Then we can divide s into $uvxyz$ as in the figure.
  - The lower occurrence of R generates x, while the upper occurrence of R generates vxy.

The proof (2)

- Then we can divide s into $uvxyz$ as in the figure.
  - The lower occurrence of R generates x, while the upper occurrence of R generates vxy.
  - By replacing the upper occurrence with lower occurrence or vise versa, we can show that uv^ixy^iz belongs to A.

The proof (2)

- Then we can divide s into $uvxyz$ as in the figure.
  - The lower occurrence of R generates x, while the upper occurrence of R generates vxy.
  - By replacing the upper occurrence with lower occurrence or vise versa, we can show that uv^ixy^iz belongs to A.
- We're not done.

The proof (2)

- Then we can divide s into $uvxyz$ as in the figure.
  - The lower occurrence of R generates x, while the upper occurrence of R generates vxy.
  - By replacing the upper occurrence with lower occurrence or vise versa, we can show that uv^ixy^iz belongs to A.
- We're not done. We have to prove other 2 properties:

The proof (2)

- Then we can divide s into $uvxyz$ as in the figure.
  - The lower occurrence of R generates x, while the upper occurrence of R generates vxy.
  - By replacing the upper occurrence with lower occurrence or vise versa, we can show that uv^ixy^iz belongs to A.
- We're not done. We have to prove other 2 properties: (2) $|vy| > 0$ and

The proof (2)

- Then we can divide s into $uvxyz$ as in the figure.
  - The lower occurrence of R generates x, while the upper occurrence of R generates vxy.
  - By replacing the upper occurrence with lower occurrence or vise versa, we can show that uv^ixy^iz belongs to A.
- We're not done. We have to prove other 2 properties: (2) $|vy| > 0$ and (3) $|vxy| \leq p$.

The proof (2)

- Then we can divide s into $uvxyz$ as in the figure.
  - The lower occurrence of R generates x, while the upper occurrence of R generates vxy.
  - By replacing the upper occurrence with lower occurrence or vise versa, we can show that uv^ixy^iz belongs to A.
- We're not done. We have to prove other 2 properties: (2) $|vy| > 0$ and (3) $|vxy| \leq p$.
  - For (2), it can be the case that between the first R and the second R, nothing gets generated.

The proof (3): using minimality

- The key to get (2) is to use minimality. Instead of choosing any parse tree τ that generates s, we choose the one with minimum number of nodes.
    - What does this mean?

The proof (3): using minimality

- The key to get (2) is to use minimality. Instead of choosing any parse tree τ that generates s, we choose the one with minimum number of nodes.
  - What does this mean?
  - Now consider $uv^0xy^0z = uxz$.

The proof (3): using minimality

- The key to get (2) is to use minimality. Instead of choosing any parse tree τ that generates s, we choose the one with minimum number of nodes.
  - What does this mean?
  - Now consider $uv^0xy^0z = uxz$. If $|vy| = 0$, i.e., prop (2) fails,

The proof (3): using minimality

- The key to get (2) is to use minimality. Instead of choosing any parse tree τ that generates s, we choose the one with minimum number of nodes.
  - What does this mean?
  - Now consider $uv^0xy^0z = uxz$. If $|vy| = 0$, i.e., prop (2) fails, we have that $uxz = uvxyz = s$.

The proof (3): using minimality

- The key to get (2) is to use minimality. Instead of choosing any parse tree τ that generates s, we choose the one with minimum number of nodes.
  - What does this mean?
  - Now consider $uv^0xy^0z = uxz$. If $|vy| = 0$, i.e., prop (2) fails, we have that $uxz = uvxyz = s$.
  - Note that the tree τ' that generates uxz has fewer number of nodes than τ. (why?)

The proof (3): using minimality

- The key to get (2) is to use minimality. Instead of choosing any parse tree τ that generates s, we choose the one with minimum number of nodes.
  - What does this mean?
  - Now consider $uv^0xy^0z = uxz$. If $|vy| = 0$, i.e., prop (2) fails, we have that $uxz = uvxyz = s$.
  - Note that the tree τ' that generates uxz has fewer number of nodes than τ. (why?) But it also generates s;

The proof (3): using minimality

- The key to get (2) is to use minimality. Instead of choosing any parse tree τ that generates s, we choose the one with minimum number of nodes.
  - What does this mean?
  - Now consider $uv^0xy^0z = uxz$. If $|vy| = 0$, i.e., prop (2) fails, we have that $uxz = uvxyz = s$.
  - Note that the tree τ' that generates uxz has fewer number of nodes than τ. (why?) But it also generates s; this **contradicts** the assumption that τ is the tree with minimum number of nodes.

The proof (3): using minimality

- The key to get (2) is to use minimality. Instead of choosing any parse tree τ that generates s, we choose the one with minimum number of nodes.
  - What does this mean?
  - Now consider $uv^0xy^0z = uxz$. If $|vy| = 0$, i.e., prop (2) fails, we have that $uxz = uvxyz = s$.
  - Note that the tree τ' that generates uxz has fewer number of nodes than τ. (why?) But it also generates s; this **contradicts** the assumption that τ is the tree with minimum number of nodes.
  - Thus, $uxz \neq uvxyz$, and $|vy| > 0$.

The proof (4)

- To prove (3), we use the same idea.

The proof (4)

- To prove (3), we use the same idea.
- We pick the occurrences of R carefully:

The proof (4)

- To prove (3), we use the same idea.
- We pick the occurrences of R carefully: we pick the **lowest** two occurrences of R.

The proof (4)

- To prove (3), we use the same idea.
- We pick the occurrences of R carefully: we pick the **lowest** two occurrences of R.
- HW: show that this implies (3).

Equivalence

Theorem 2

A language is context-free if and only if some pushdown automaton recognizes it.

Recall two directions

- **Only-if:** If a language is context-free, it is recognized by some pushdown automaton.

Recall two directions

- **Only-if:** If a language is context-free, it is recognized by some pushdown automaton.
    - Given a CFG G, construct a PDA P that recognizes the language generated by G.

Recall two directions

- **Only-if:** If a language is context-free, it is recognized by some pushdown automaton.
    - Given a CFG G, construct a PDA P that recognizes the language generated by G. **DONE.**

Recall two directions

- **Only-if:** If a language is context-free, it is recognized by some pushdown automaton.
  - Given a CFG G, construct a PDA P that recognizes the language generated by G. **DONE.**
- **If:** A language is context-free if it is recognized by some pushdown automaton.

Recall two directions

- **Only-if:** If a language is context-free, it is recognized by some pushdown automaton.
  - Given a CFG G, construct a PDA P that recognizes the language generated by G. **DONE.**
- **If:** A language is context-free if it is recognized by some pushdown automaton.
  - Given a PDA P, construct a CFG G that generates a language recognized by P.

Recall two directions

- **Only-if:** If a language is context-free, it is recognized by some pushdown automaton.
  - Given a CFG G, construct a PDA P that recognizes the language generated by G. **DONE.**
- **If:** A language is context-free if it is recognized by some pushdown automaton.
  - Given a PDA P, construct a CFG G that generates a language recognized by P. **SKIPPED.**

Skipped...

The second part is quite technical, so we decide to skip the proof of the second part.

Models of computation

- **Finite automata and regular expressions.**
  - Devices with small, limited memory.
- **Push-down automata and context-free languages**
  - Devices with unlimited memory, but have restricted access.

Turing Machines

Turing Machines

- Proposed by Alan Turing in 1936.

Turing Machines

- Proposed by Alan Turing in 1936.
- A finite automaton with an unlimited memory with unrestricted access.

Turing Machines

- Proposed by Alan Turing in 1936.
- A finite automaton with an unlimited memory with unrestricted access.
- Can perform any tasks that a computer can. (we'll see)

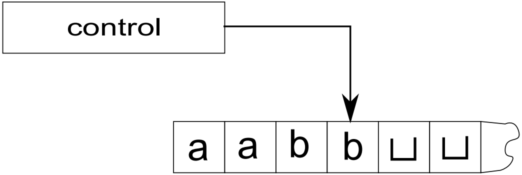
Turing Machines

- Proposed by Alan Turing in 1936.
- A finite automaton with an unlimited memory with unrestricted access.
- Can perform any tasks that a computer can. (we'll see)
- However, there are problems that TM can't solve. These problems are beyond the limit of computation.

Components

- An infinite **tape**.
- A tape head that can
  - **read and write** to the tape, and
  - **move** around the tape.

Schematic



How Turing machines work

- The tape initialy contains an input string.

How Turing machines work

- The tape initialy contains an input string.
- The rest of the tape is blank (denoted by $\sqcup$).

How Turing machines work

- The tape initialy contains an input string.
- The rest of the tape is blank (denoted by $\sqcup$).
- The machine reads a symbol from of the tape where its head is at.

How Turing machines work

- The tape initialy contains an input string.
- The rest of the tape is blank (denoted by $\sqcup$).
- The machine reads a symbol from of the tape where its head is at.
- It can write a symbol back and move left or right.

How Turing machines work

- The tape initialy contains an input string.
- The rest of the tape is blank (denoted by $\sqcup$).
- The machine reads a symbol from of the tape where its head is at.
- It can write a symbol back and move left or right.
- At the end of the computation, the machine outputs accept or reject, by entering accept state of reject state. (After changing, it halts.)

How Turing machines work

- The tape initialy contains an input string.
- The rest of the tape is blank (denoted by $\sqcup$).
- The machine reads a symbol from of the tape where its head is at.
- It can write a symbol back and move left or right.
- At the end of the computation, the machine outputs accept or reject, by entering accept state of reject state. (After changing, it halts.)
- It can go on forever (not entering any accept or reject states).

Example: M_1

- We'll design a TM that recognizes

$$B = \{w\#w \mid w \in \{0,1\}^*\}.$$

Example: M_1 — strategy

- M_1 works by comparing two copies of w.
- M_1 compares two symbols on the corresponding positions.
- It write marks on the tape to keep track of the position.

Example: M_1 — snapshots

$\textcolor{red}{0}\ 1\ 1\ 0\ 0\ 0\ \#\ 0\ 1\ 1\ 0\ 0\ 0\ \sqcup$

Example: M_1 — snapshots

$\color{red}{0}\ 1\ 1\ 0\ 0\ 0\ \#\ 0\ 1\ 1\ 0\ 0\ 0\ \sqcup$

$\mathtt{x}\ \color{red}{1}\ 1\ 0\ 0\ 0\ \#\ 0\ 1\ 1\ 0\ 0\ 0\ \sqcup$

Example: M_1 — snapshots

```
0 1 1 0 0 0 # 0 1 1 0 0 0 ⊔
x 1 1 0 0 0 # 0 1 1 0 0 0 ⊔
x 1 1 0 0 0 # 0 1 1 0 0 0 ⊔
```

Example: M_1 — snapshots

$$
\begin{array}{cccccccccccccc}
\color{red}{0} & 1 & 1 & 0 & 0 & 0 & \# & 0 & 1 & 1 & 0 & 0 & 0 & \sqcup \\
\mathtt{x} & \color{red}{1} & 1 & 0 & 0 & 0 & \# & 0 & 1 & 1 & 0 & 0 & 0 & \sqcup \\
\mathtt{x} & 1 & 1 & 0 & 0 & 0 & \# & \color{red}{0} & 1 & 1 & 0 & 0 & 0 & \sqcup \\
\mathtt{x} & 1 & 1 & 0 & 0 & 0 & \# & \color{red}{\mathtt{x}} & 1 & 1 & 0 & 0 & 0 & \sqcup
\end{array}
$$

Example: M_1 — snapshots

$$
\begin{array}{cccccccccccccc}
\color{red}{0} & 1 & 1 & 0 & 0 & 0 & \# & 0 & 1 & 1 & 0 & 0 & 0 & \sqcup \\
\mathtt{x} & \color{red}{1} & 1 & 0 & 0 & 0 & \# & 0 & 1 & 1 & 0 & 0 & 0 & \sqcup \\
\mathtt{x} & 1 & 1 & 0 & 0 & 0 & \# & \color{red}{0} & 1 & 1 & 0 & 0 & 0 & \sqcup \\
\mathtt{x} & 1 & 1 & 0 & 0 & 0 & \# & \color{red}{\mathtt{x}} & 1 & 1 & 0 & 0 & 0 & \sqcup \\
\color{red}{\mathtt{x}} & 1 & 1 & 0 & 0 & 0 & \# & \mathtt{x} & 1 & 1 & 0 & 0 & 0 & \sqcup
\end{array}
$$

Example: M_1 — snapshots

$\color{red}{0}$ 1 1 0 0 0 # 0 1 1 0 0 0 ⊔

x $\color{red}{1}$ 1 0 0 0 # 0 1 1 0 0 0 ⊔

x 1 1 0 0 0 # $\color{red}{0}$ 1 1 0 0 0 ⊔

x 1 1 0 0 0 # $\color{red}{x}$ 1 1 0 0 0 ⊔

$\color{red}{x}$ 1 1 0 0 0 # x 1 1 0 0 0 ⊔

x $\color{red}{1}$ 1 0 0 0 # x 1 1 0 0 0 ⊔

Example: M_1 — snapshots

0	1	1	0	0	0	#	0	1	1	0	0	0	␣
x	1	1	0	0	0	#	0	1	1	0	0	0	␣
x	1	1	0	0	0	#	0	1	1	0	0	0	␣
x	1	1	0	0	0	#	x	1	1	0	0	0	␣
x	1	1	0	0	0	#	x	1	1	0	0	0	␣
x	1	1	0	0	0	#	x	1	1	0	0	0	␣
x	x	1	0	0	0	#	x	1	1	0	0	0	␣

Example: M_1 — snapshots

0	1	1	0	0	0	#	0	1	1	0	0	0	⊔
x	1	1	0	0	0	#	0	1	1	0	0	0	⊔
x	1	1	0	0	0	#	0	1	1	0	0	0	⊔
x	1	1	0	0	0	#	x	1	1	0	0	0	⊔
x	1	1	0	0	0	#	x	1	1	0	0	0	⊔
x	1	1	0	0	0	#	x	1	1	0	0	0	⊔
x	x	1	0	0	0	#	x	1	1	0	0	0	⊔
x	x	1	0	0	0	#	x	x	1	0	0	0	⊔

Example: M_1 — snapshots

```
0 1 1 0 0 0 # 0 1 1 0 0 0 ⊔
x 1 1 0 0 0 # 0 1 1 0 0 0 ⊔
x 1 1 0 0 0 # 0 1 1 0 0 0 ⊔
x 1 1 0 0 0 # x 1 1 0 0 0 ⊔
x 1 1 0 0 0 # x 1 1 0 0 0 ⊔
x 1 1 0 0 0 # x 1 1 0 0 0 ⊔
x x 1 0 0 0 # x 1 1 0 0 0 ⊔
x x 1 0 0 0 # x x 1 0 0 0 ⊔
x x x x x x # x x x x x x ⊔   accept!
```

Example: M_1 — algorithm

$M_1 =$ "On input string w:

1. Zig-zag across the tape to corresponding positions on either side of the # symbol to check if they contains the same symbol.

Example: M_1 — algorithm

M_1 = "On input string w:

1. Zig-zag across the tape to corresponding positions on either side of the # symbol to check if they contains the same symbol. If they do not or there is no #, reject.

Example: M_1 — algorithm

M_1 = “On input string w:

1. Zig-zag across the tape to corresponding positions on either side of the # symbol to check if they contains the same symbol. If they do not or there is no #, reject. Mark these symbols to keep track of the current position.

Example: M_1 — algorithm

M_1 = "On input string w:

1. Zig-zag across the tape to corresponding positions on either side of the # symbol to check if they contains the same symbol. If they do not or there is no #, reject. Mark these symbols to keep track of the current position.
2. After all symbols on the left of # have been marked, check if there're other unmarked symbols on the right of #, if there's any, reject; otherwise accept."

Formal definition of a Turing Machine

- Again, the important part is the definition of the transition function.

Formal definition of a Turing Machine

- Again, the important part is the definition of the transition function.
- The machine look at the tape symbol and consider its current state, then makes a move by writing some symbol on the tape and moving its head left or right.

Formal definition of a Turing Machine

- Again, the important part is the definition of the transition function.
- The machine look at the tape symbol and consider its current state, then makes a move by writing some symbol on the tape and moving its head left or right.
- Thus,
  - **Input:** current state and the symbol on the tape

Formal definition of a Turing Machine

- Again, the important part is the definition of the transition function.
- The machine look at the tape symbol and consider its current state, then makes a move by writing some symbol on the tape and moving its head left or right.
- Thus,
    - **Input:** current state and the symbol on the tape
    - **Output:** next state, a symbol to be written to the tape, and the new state.

Formal definition of a Turing Machine

- Again, the important part is the definition of the transition function.
- The machine look at the tape symbol and consider its current state, then makes a move by writing some symbol on the tape and moving its head left or right.
- Thus,
  - **Input:** current state and the symbol on the tape
  - **Output:** next state, a symbol to be written to the tape, and the new state.

Formal definition of a Turing Machine

- Again, the important part is the definition of the transition function.
- The machine look at the tape symbol and consider its current state, then makes a move by writing some symbol on the tape and moving its head left or right.
- Thus,
  - **Input:** current state and the symbol on the tape
  - **Output:** next state, a symbol to be written to the tape, and the new state.
- So, δ is in the form: $Q \times \Gamma \rightarrow Q \times \Gamma \times \{\mathrm{L}, \mathrm{R}\}$

Formal definition of a Turing Machine

- Again, the important part is the definition of the transition function.
- The machine look at the tape symbol and consider its current state, then makes a move by writing some symbol on the tape and moving its head left or right.
- Thus,
  - **Input:** current state and the symbol on the tape
  - **Output:** next state, a symbol to be written to the tape, and the new state.
- So, δ is in the form: $Q \times \Gamma \rightarrow Q \times \Gamma \times \{\mathrm{L}, \mathrm{R}\}$
- E.g., if $\delta(q, a) = (r, b, \mathrm{L})$, then if the machine is in state q and reads a, it will change its state to r, write b to the tape and move to the left.

Definition

Definition (Turing Machine)

A **Turing machine** is a 7-tuple, $(Q, \Sigma, \Gamma, \delta, q_0, q_{accept}, q_{reject})$, where Q, Σ, Γ are finite sets and

1. Q is the set of states,
2. Σ is the input alphabet not containing the **blank symbol** $\sqcup$,
3. Γ is the tape alphabet, where $\sqcup \in \Gamma$ and $\Sigma \subset \Gamma$,
4. $\delta : Q \times \Gamma \rightarrow Q \times \Gamma \times \{\mathrm{L}, \mathrm{R}\}$ is the transition function,
5. $q_0 \in Q$ is the start state,
6. $q_{accept} \in Q$ is the accept state, and
7. $q_{reject} \in Q$ is the reject state, where $q_{accept} \neq q_{reject}$.

Differences between TM and FA

Differences between TM and FA

- A Turing machine can read from the tape and write to it.

Differences between TM and FA

- A Turing machine can read from the tape and write to it.
- The read-write head can move both to the left and to the right.

Differences between TM and FA

- A Turing machine can read from the tape and write to it.
- The read-write head can move both to the left and to the right.
- The tape is infinite.

Differences between TM and FA

- A Turing machine can read from the tape and write to it.
- The read-write head can move both to the left and to the right.
- The tape is infinite.
- The special states for accepting and rejecting take effect immediately.

- A collection of strings that a Turing machine M accepts is the language of M, denoted by $L(M)$.

- A collection of strings that a Turing machine M accepts is the language of M, denoted by $L(M)$.
- We say that M recognizes $L(M)$.

- A collection of strings that a Turing machine M accepts is the language of M, denoted by $L(M)$.
- We say that M recognizes $L(M)$.

Definition

A language is called **Turing-recognizable** if some Turing machine recognizes it.

Output of a TM

- The output of a TM can be accept, reject, or loop.

Output of a TM

- The output of a TM can be accept, reject, or loop.
- A Turing machine may not accept or reject a string.

Output of a TM

- The output of a TM can be accept, reject, or loop.
- A Turing machine may not accept or reject a string.
- We are interested particularly in TM thats halts (does not loop).

Output of a TM

- The output of a TM can be accept, reject, or loop.
- A Turing machine may not accept or reject a string.
- We are interested particularly in TM thats halts (does not loop). We call them **decider**.

Output of a TM

- The output of a TM can be accept, reject, or loop.
- A Turing machine may not accept or reject a string.
- We are interested particularly in TM thats halts (does not loop). We call them **decider**.
- A decider that recognizes some language also is said to **decides** that langauge.

Definition

A language is called **Turing-decidable** or **decidable** if some Turing machine decides it.

Example: M_2

Design M_2 that decides the language $A = \{0^{2^n} \mid n \geq 0\}$.

M_2's algorithm

On input string w:

1. Sweep left to right across the tape, crossing off every other 0.

M_2's algorithm

On input string w:

1. Sweep left to right across the tape, crossing off every other 0.
2. If in state 1 the tape contained a single 0, accept.

M_2's algorithm

On input string w:

1. Sweep left to right across the tape, crossing off every other 0.
2. If in state 1 the tape contained a single 0, accept.
3. If in state 1 the tape contained more than a single 0 and the number of 0 was odd, reject.

M_2's algorithm

On input string w:

1. Sweep left to right across the tape, crossing off every other 0.
2. If in state 1 the tape contained a single 0, accept.
3. If in state 1 the tape contained more than a single 0 and the number of 0 was odd, reject.
4. Return the head to the left-hand end of the tape.
5. Go to state 1.

Configuration

- At any point of the computation, the TM can be in some state, and at some position on the tape.

Configuration

- At any point of the computation, the TM can be in some state, and at some position on the tape.
- A **configuration** of the Turing machine, "the current computing status" can be defined with the current state, the current position of the tape head, and the content of the tape.

Configuration

- At any point of the computation, the TM can be in some state, and at some position on the tape.
- A **configuration** of the Turing machine, "the current computing status" can be defined with the current state, the current position of the tape head, and the content of the tape.
- We usually write configurate as: $u\ q\ v$, where q is the state, uv is the current content of the tape, and the TM is at the first symbol of v.

Example execution of M_2