INF2080

Repetition

Daniel Lupp

Universitetet i Oslo

9th March 2018

Department of Informatics

University of Oslo

Today

- wrap-up of last week Friday
- repetition of course so far

Wrap-up: Reducibility

Idea: Convert a problem A into a second problem B in such a way that a solution for B gives us a solution for A.

Wrap-up: Reducibility

Idea: Convert a problem A into a second problem B in such a way that a solution for B gives us a solution for A.

Example: Last week's bad mathematician joke:

- A mathematician and an engineer go camping. After setting up camp, they go back to their car to get their pots, go to the river to fetch water, put the water on the fire to boil.

Wrap-up: Reducibility

Idea: Convert a problem A into a second problem B in such a way that a solution for B gives us a solution for A.

Example: Last week's bad mathematician joke:

- A mathematician and an engineer go camping. After setting up camp, they go back to their car to get their pots, go to the river to fetch water, put the water on the fire to boil.
- the next morning, they both need to boil water. The engineer fills up one pot with water and begins heating it up.

Wrap-up: Reducibility

Idea: Convert a problem A into a second problem B in such a way that a solution for B gives us a solution for A.

Example: Last week's bad mathematician joke:

- A mathematician and an engineer go camping. After setting up camp, they go back to their car to get their pots, go to the river to fetch water, put the water on the fire to boil.
- the next morning, they both need to boil water. The engineer fills up one pot with water and begins heating it up.
- The mathematician reduces the current problem ("boil water after a night camping") to a problem with a known solution.

Wrap-up: Reducibility

Idea: Convert a problem A into a second problem B in such a way that a solution for B gives us a solution for A.
Example: Last week's bad mathematician joke:

- A mathematician and an engineer go camping. After setting up camp, they go back to their car to get their pots, go to the river to fetch water, put the water on the fire to boil.
- the next morning, they both need to boil water. The engineer fills up one pot with water and begins heating it up.
- The mathematician reduces the current problem ("boil water after a night camping") to a problem with a known solution. The mathematician (1) brings the pot back to the car,

Wrap-up: Reducibility

Idea: Convert a problem A into a second problem B in such a way that a solution for B gives us a solution for A.

Example: Last week's bad mathematician joke:

- A mathematician and an engineer go camping. After setting up camp, they go back to their car to get their pots, go to the river to fetch water, put the water on the fire to boil.
- the next morning, they both need to boil water. The engineer fills up one pot with water and begins heating it up.
- The mathematician reduces the current problem ("boil water after a night camping") to a problem with a known solution. The mathematician (1) brings the pot back to the car, (2) goes back to camp.

Wrap-up: Reducibility

Idea: Convert a problem A into a second problem B in such a way that a solution for B gives us a solution for A.
Example: Last week's bad mathematician joke:

- A mathematician and an engineer go camping. After setting up camp, they go back to their car to get their pots, go to the river to fetch water, put the water on the fire to boil.
- the next morning, they both need to boil water. The engineer fills up one pot with water and begins heating it up.
- The mathematician reduces the current problem ("boil water after a night camping") to a problem with a known solution. The mathematician (1) brings the pot back to the car, (2) goes back to camp. Thus the problem has been reduced to the problem solved the day before: how to get boiling water when the pot is in the car.

Wrap-up: Reducibility

We used this to show various decidability and undecidability results, e.g.,

- $HALT_{TM} = \{\langle M, w \rangle \mid M \text{ is a TM that halts on input } w\}$ is undecidable (reduction from A_{TM})

Wrap-up: Reducibility

We used this to show various decidability and undecidability results, e.g.,

- $HALT_{TM} = \{\langle M, w\rangle \mid M \text{ is a TM that halts on input } w\}$ is undecidable (reduction from A_{TM})
- $E_{TM} = \{\langle M\rangle \mid M \text{ is a TM accepts no input}\}$ is undecidable (reduction from A_{TM})

Wrap-up: Reducibility

We used this to show various decidability and undecidability results, e.g.,

- $HALT_{TM} = \{\langle M, w\rangle \mid M \text{ is a TM that halts on input } w\}$ is undecidable (reduction from A_{TM})
- $E_{TM} = \{\langle M\rangle \mid M \text{ is a TM accepts no input}\}$ is undecidable (reduction from A_{TM})
- $REGULAR_{TM} = \{\langle M\rangle \mid M \text{ is a TM that accepts a regular language}\}$ is undecidable (reduction from A_{TM})

Wrap-up: Reducibility

We used this to show various decidability and undecidability results, e.g.,

- $HALT_{TM} = \{\langle M, w\rangle \mid M$ is a TM that halts on input $w\}$ is undecidable (reduction from A_{TM})
- $E_{TM} = \{\langle M\rangle \mid M$ is a TM accepts no input$\}$ is undecidable (reduction from A_{TM})
- $REGULAR_{TM} = \{\langle M\rangle \mid M$ is a TM that accepts a regular language$\}$ is undecidable (reduction from A_{TM})
- Rice's theorem: checking any nontrivial property of a Turing machine (if it's regular, context-free, etc.) is undecidable!

Wrap-up: Reducibility

We used this to show various decidability and undecidability results, e.g.,

- $HALT_{TM} = \{\langle M, w \rangle \mid M \text{ is a TM that halts on input } w\}$ is undecidable (reduction from A_{TM})
- $E_{TM} = \{\langle M \rangle \mid M \text{ is a TM accepts no input}\}$ is undecidable (reduction from A_{TM})
- $REGULAR_{TM} = \{\langle M \rangle \mid M \text{ is a TM that accepts a regular language}\}$ is undecidable (reduction from A_{TM})
- Rice's theorem: checking any nontrivial property of a Turing machine (if it's regular, context-free, etc.) is undecidable!
- ...

Wrap-up: Reducibility

We formalized reducibility as follows:

Definition

Language A is *mapping reducible* to language B, written $A \leq_m B$, if there exists a computable function $f : \Sigma^* \rightarrow \Sigma^*$ such that for every w

$$w \in A \iff f(w) \in B$$

Wrap-up: Reducibility

We formalized reducibility as follows:

Definition

Language A is *mapping reducible* to language B, written $A \leq_m B$, if there exists a computable function $f : \Sigma^* \rightarrow \Sigma^*$ such that for every w

$$w \in A \iff f(w) \in B$$

Recall: a function $f : \Sigma^* \rightarrow \Sigma^*$ is computable if there exists a Turing machine M that for every input w halts with just $f(w)$ on its tape.

Wrap-up: Reducibility

Definition

Language A is *mapping reducible* to language B, written $A \leq_m B$, if there exists a computable function $f : \Sigma^* \to \Sigma^*$ such that for every w

$$w \in A \iff f(w) \in B$$

By this definition: $A \leq_m B \iff \overline{A} \leq_m \overline{B}$ (useful tool we will use soon).

Wrap-up: Reducibility

Theorem

If $A \leq_m B$ and B is decidable [Turing-recognizable], then A is decidable [Turing-recognizable].

Wrap-up: Reducibility

Theorem

If $A \leq_m B$ and B is decidable [Turing-recognizable], then A is decidable [Turing-recognizable].

Theorem

If $A \leq_m B$ and A is undecidable [non-Turing-recognizable], then B is undecidable [non-Turing-recognizable].

Wrap-up: Reducibility

We can use this to show:

Theorem

The language $EQ_{TM} = \{\langle M_1, M_2\rangle \mid M_1, M_2$ are TMs with $L(M_1) = L(M_2)\}$ is neither Turing-recognizable nor co-Turing-recognizable.

Proof: Let's show that EQ_{TM} is not Turing recognizable. We show a mapping reduction from $\overline{A_{TM}}$, i.e., $\overline{A_{TM}} \leq_m EQ_{TM}$.

Wrap-up: Reducibility

We can use this to show:

Theorem

The language $EQ_{TM} = \{\langle M_1, M_2 \rangle \mid M_1, M_2$ are TMs with $L(M_1) = L(M_2)\}$ is neither Turing-recognizable nor co-Turing-recognizable.

Proof: Let's show that EQ_{TM} is not Turing recognizable. We show a mapping reduction from $\overline{A_{TM}}$, i.e., $\overline{A_{TM}} \leq_m EQ_{TM}$. This is the same as showing $A_{TM} \leq_m \overline{EQ_{TM}}$.

Wrap-up: Reducibility

We can use this to show:

Theorem

The language $EQ_{TM} = \{\langle M_1, M_2\rangle \mid M_1, M_2$ are TMs with $L(M_1) = L(M_2)\}$ is neither Turing-recognizable nor co-Turing-recognizable.

Proof: Let's show that EQ_{TM} is not Turing recognizable. We show a mapping reduction from $\overline{A_{TM}}$, i.e., $\overline{A_{TM}} \leq_m EQ_{TM}$. This is the same as showing $A_{TM} \leq_m \overline{EQ_{TM}}$. The computable function is described by the following Turing machine:

$F =$ on input $\langle M, w\rangle$:

1. Construct the following two machines M_1 and M_2:
   M_1 : on any input, *reject*
   M_2 : on any input, run M on w. If it accepts, *accept*.
2. Output $\langle M_1, M_2\rangle$.

Wrap-up: Reducibility

Theorem

The language $EQ_{TM} = \{\langle M_1, M_2\rangle \mid M_1, M_2$ *are TMs with* $L(M_1) = L(M_2)\}$ *is neither Turing-recognizable nor co-Turing-recognizable.*

Proof:

$F =$ on input $\langle M, w\rangle$:

1. Construct the following two machines M_1 and M_2:
   M_1 : on any input, *reject*
   M_2 : on any input, run M on w. If it accepts, *accept*.
2. Output $\langle M_1, M_2\rangle$.

Wrap-up: Reducibility

Theorem

The language $EQ_{TM} = \{\langle M_1, M_2 \rangle \mid M_1, M_2$ *are TMs with* $L(M_1) = L(M_2)\}$ *is neither Turing-recognizable nor co-Turing-recognizable.*

Proof:

$F =$ on input $\langle M, w \rangle$:

1. Construct the following two machines M_1 and M_2:
   M_1 : on any input, *reject*
   M_2 : on any input, run M on w. If it accepts, *accept*.
2. Output $\langle M_1, M_2 \rangle$.

- $L(M_2) = \emptyset$ if M does not accept w.

Wrap-up: Reducibility

Theorem

The language $EQ_{TM} = \{\langle M_1, M_2\rangle \mid M_1, M_2$ *are TMs with* $L(M_1) = L(M_2)\}$ *is neither Turing-recognizable nor co-Turing-recognizable.*

Proof:

$F =$ on input $\langle M, w\rangle$:

1. Construct the following two machines M_1 and M_2:
   M_1 : on any input, *reject*
   M_2 : on any input, run M on w. If it accepts, *accept*.
2. Output $\langle M_1, M_2\rangle$.

- $L(M_2) = \emptyset$ if M does not accept w.
- $L(M_2) = \Sigma^*$ if M accepts w.

Wrap-up: Reducibility

Theorem

The language $EQ_{TM} = \{\langle M_1, M_2\rangle \mid M_1, M_2$ *are TMs with* $L(M_1) = L(M_2)\}$ *is neither Turing-recognizable nor co-Turing-recognizable.*

Proof:

$F =$ on input $\langle M, w\rangle$:

1. Construct the following two machines M_1 and M_2:
   M_1 : on any input, *reject*
   M_2 : on any input, run M on w. If it accepts, *accept*.
2. Output $\langle M_1, M_2\rangle$.

- $L(M_2) = \emptyset$ if M does not accept w.
- $L(M_2) = \Sigma^*$ if M accepts w.
- Thus, $L(M_1) \neq L(M_2)$ iff M accepts w, and $A_{TM} \leq_m \overline{EQ_{TM}}$

Wrap-up: Reducibility

Theorem

The language $EQ_{TM} = \{\langle M_1, M_2\rangle \mid M_1, M_2$ *are TMs with* $L(M_1) = L(M_2)\}$ *is neither Turing-recognizable nor co-Turing-recognizable.*

Proof: Next show that EQ_{TM} is not co-Turing recognizable. We show a mapping reduction from $\overline{A_{TM}}$, i.e., $\overline{A_{TM}} \leq_m \overline{EQ_{TM}}$.

Wrap-up: Reducibility

Theorem

The language $EQ_{TM} = \{\langle M_1, M_2\rangle \mid M_1, M_2$ *are TMs with* $L(M_1) = L(M_2)\}$ *is neither Turing-recognizable nor co-Turing-recognizable.*

Proof: Next show that EQ_{TM} is not co-Turing recognizable. We show a mapping reduction from $\overline{A_{TM}}$, i.e., $\overline{A_{TM}} \leq_m \overline{EQ_{TM}}$. This is the same as showing $A_{TM} \leq_m EQ_{TM}$.

Wrap-up: Reducibility

Theorem

The language $EQ_{TM} = \{\langle M_1, M_2\rangle \mid M_1, M_2$ *are TMs with* $L(M_1) = L(M_2)\}$ *is neither Turing-recognizable nor co-Turing-recognizable.*

Proof: Next show that EQ_{TM} is not co-Turing recognizable. We show a mapping reduction from $\overline{A_{TM}}$, i.e., $\overline{A_{TM}} \leq_m \overline{EQ_{TM}}$. This is the same as showing $A_{TM} \leq_m EQ_{TM}$. The computable function is described by the following Turing machine:

$F =$ on input $\langle M, w\rangle$:

1. Construct the following two machines M_1 and M_2:
   M_1 : on any input, *accept*
   M_2 : on any input, run M on w. If it accepts, *accept*.
2. Output $\langle M_1, M_2\rangle$.

Wrap-up: Reducibility

Theorem

The language $EQ_{TM} = \{\langle M_1, M_2\rangle \mid M_1, M_2$ *are TMs with* $L(M_1) = L(M_2)\}$ *is neither Turing-recognizable nor co-Turing-recognizable.*

Proof:

$F =$ on input $\langle M, w\rangle$:

1. Construct the following two machines M_1 and M_2:
   M_1 : on any input, *accept*
   M_2 : on any input, run M on w. If it accepts, *accept*.
2. Output $\langle M_1, M_2\rangle$.

Wrap-up: Reducibility

Theorem

The language $EQ_{TM} = \{\langle M_1, M_2\rangle \mid M_1, M_2$ *are TMs with* $L(M_1) = L(M_2)\}$ *is neither Turing-recognizable nor co-Turing-recognizable.*

Proof:

$F =$ on input $\langle M, w\rangle$:

1. Construct the following two machines M_1 and M_2:
   M_1 : on any input, *accept*
   M_2 : on any input, run M on w. If it accepts, *accept*.
2. Output $\langle M_1, M_2\rangle$.

- $L(M_2) = \emptyset$ if M does not accept w

Wrap-up: Reducibility

Theorem

The language $EQ_{TM} = \{\langle M_1, M_2 \rangle \mid M_1, M_2$ *are TMs with* $L(M_1) = L(M_2)\}$ *is neither Turing-recognizable nor co-Turing-recognizable.*

Proof:

$F =$ on input $\langle M, w \rangle$:

1. Construct the following two machines M_1 and M_2:
   M_1 : on any input, *accept*
   M_2 : on any input, run M on w. If it accepts, *accept*.
2. Output $\langle M_1, M_2 \rangle$.

- $L(M_2) = \emptyset$ if M does not accept w
- $L(M_2) = \Sigma^*$ if M accepts w.

Wrap-up: Reducibility

Theorem

The language $EQ_{TM} = \{\langle M_1, M_2\rangle \mid M_1, M_2$ *are TMs with* $L(M_1) = L(M_2)\}$ *is neither Turing-recognizable nor co-Turing-recognizable.*

Proof:

$F =$ on input $\langle M, w\rangle$:

1. Construct the following two machines M_1 and M_2:
   M_1 : on any input, *accept*
   M_2 : on any input, run M on w. If it accepts, *accept*.
2. Output $\langle M_1, M_2\rangle$.

- $L(M_2) = \emptyset$ if M does not accept w
- $L(M_2) = \Sigma^*$ if M accepts w.
- Thus, $L(M_1) = L(M_2)$ iff M accepts w, and $A_{TM} \leq_m EQ_{TM}$

Wrap-up: Reducibility

Let's consider the implications:

- We have seen that Turing machines capture the expressivity of *any* computational model that has unlimited access to infinite memory that is allowed to do finite work per step

Wrap-up: Reducibility

Let's consider the implications:

- We have seen that Turing machines capture the expressivity of *any* computational model that has unlimited access to infinite memory that is allowed to do finite work per step
- There exist languages that are not algorithmically solvable, i.e., membership and non-membership determined after a finite number of steps (undecidable, e.g., $HALT_{TM}$)

Wrap-up: Reducibility

Let's consider the implications:

- We have seen that Turing machines capture the expressivity of *any* computational model that has unlimited access to infinite memory that is allowed to do finite work per step
- There exist languages that are not algorithmically solvable, i.e., membership and non-membership determined after a finite number of steps (undecidable, e.g., $HALT_{TM}$)
- There exist languages that are not recognizable, i.e., no Turing machine can check membership after finite steps (non-Turing-recognizable, e.g., $\overline{A_{TM}}$)

Wrap-up: Reducibility

Let's consider the implications:

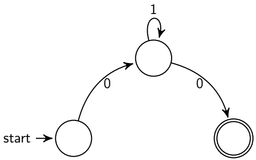
- We have seen that Turing machines capture the expressivity of *any* computational model that has unlimited access to infinite memory that is allowed to do finite work per step
- There exist languages that are not algorithmically solvable, i.e., membership and non-membership determined after a finite number of steps (undecidable, e.g., $HALT_{TM}$)
- There exist languages that are not recognizable, i.e., no Turing machine can check membership after finite steps (non-Turing-recognizable, e.g., $\overline{A_{TM}}$)
- There exist languages that are neither recognizable nor co-recognizable, i.e., *no such computational model* can check membership or non-membership! (e.g., $\overline{EQ_{TM}}$)

Regular Languages

- Determininstic Finite Automata (DFA): an automata with a finite number of states where for every state and input there is precisely one transition leading to another state.

Regular Languages

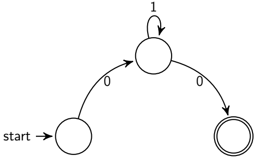
- Determininstic Finite Automata (DFA): an automata with a finite number of states where for every state and input there is precisely one transition leading to another state.



- contain a start state, possibly multiple accepting states. If after starting in the start state, parsing an input and following correct transitions the automaton ends in an accept state, the input is accepted

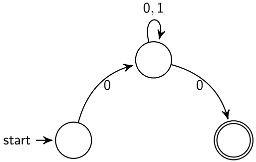
Regular Languages

- Determininstic Finite Automata (DFA): an automata with a finite number of states where for every state and input there is precisely one transition leading to another state.

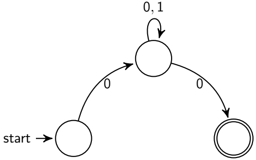


- contain a start state, possibly multiple accepting states. If after starting in the start state, parsing an input and following correct transitions the automaton ends in an accept state, the input is accepted
- The set of inputs accepted by a DFA is called a *regular language*

- We can add *nondeterminism*: given a state and a current input symbol, multiple possible following states:



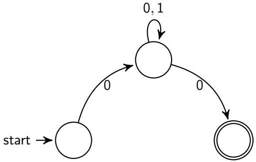
- We can add *nondeterminism*: given a state and a current input symbol, multiple possible following states:



- NFA's accept the same languages as DFA's, i.e., a language is regular iff an NFA accepts it

Regular Languages

- We can add *nondeterminism*: given a state and a current input symbol, multiple possible following states:



- NFA's accept the same languages as DFA's, i.e., a language is regular iff an NFA accepts it
- proof idea: Given an NFA N with state set Q, we define a DFA D with state set $P(Q)$, where the state $Q \in P(Q)$ in D represents that N could be in any state $q \in Q$.

Regular Languages

Another way of encoding regular languages are *regular expressions*: strings constructed from symbols from the alphabet Σ and the operations: Kleene star($*$), union ($\cup$), and concatanation.

Regular Languages

Another way of encoding regular languages are *regular expressions*: strings constructed from symbols from the alphabet Σ and the operations: Kleene star($*$), union ($\cup$), and concatanation.

- Order of operations: Kleene star binds stronger than concatanation, which binds stronger than union:
  Example: $0 \cup 10^* = (0) \cup (1(0^*))$

Regular Languages

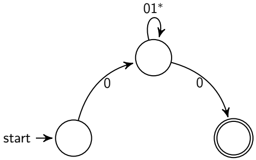
Another way of encoding regular languages are *regular expressions*: strings constructed from symbols from the alphabet Σ and the operations: Kleene star($*$), union ($\cup$), and concatanation.

- Order of operations: Kleene star binds stronger than concatanation, which binds stronger than union:
  Example: $0 \cup 10^* = (0) \cup (1(0^*))$
  $\rightarrow$ remember to use parentheses when necessary!!

Regular Languages

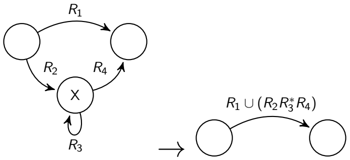
Another way of encoding regular languages are *regular expressions*: strings constructed from symbols from the alphabet Σ and the operations: Kleene star($*$), union ($\cup$), and concatanation.

- Order of operations: Kleene star binds stronger than concatanation, which binds stronger than union:
  Example: $0 \cup 10^* = (0) \cup (1(0^*))$
  $\rightarrow$ remember to use parentheses when necessary!!
- The expressivity of regular languages is precisely that of DFA/NFA. To show this, we introduced GNFA (generalized finite automata), NFA's with RE's as labels instead of symbols.



Proof idea that RE=DFA: Consider a DFA as a GNFA. Then iteratively remove nodes, and encode paths through that node in other edges:

Proof idea that RE=DFA: Consider a DFA as a GNFA. Then iteratively remove nodes, and encode paths through that node in other edges:



Pumping Lemma - Regular Languages

Lemma (Pumping Lemma)

If A is a regular language, then there is a number p, called the pumping length, where if w is a word in A of length $\geq p$ then w can be divided into three parts, $w = xyz$, such that

1. $xy^iz \in A$ *for every* $i \geq 0$,
2. $|y| > 0$,
3. $|xy| \leq p$.

Pumping Lemma - Regular Languages

Lemma (Pumping Lemma)

If A is a regular language, then there is a number p, called the pumping length, where if w is a word in A of length $\geq p$ then w can be divided into three parts, $w = xyz$, such that

1. $xy^iz \in A$ *for every* $i \geq 0$,
2. $|y| > 0$,
3. $|xy| \leq p$.

- Use the fact that regular languages only have finite memory

Pumping Lemma - Regular Languages

Lemma (Pumping Lemma)

If A is a regular language, then there is a number p, called the pumping length, where if w is a word in A of length $\geq p$ then w can be divided into three parts, $w = xyz$, such that

1. $xy^iz \in A$ *for every* $i \geq 0$,
2. $|y| > 0$,
3. $|xy| \leq p$.

- Use the fact that regular languages only have finite memory
- An automaton's memory is represented by the states, i.e., if a word is longer than the number of states (=available memory), some state must be repeated twice in the accepting path. $\rightarrow$ cycle!

Pumping Lemma - Regular Languages

Lemma (Pumping Lemma)

If A is a regular language, then there is a number p, called the pumping length, where if w is a word in A of length $\geq p$ then w can be divided into three parts, $w = xyz$, such that

1. $xy^iz \in A$ *for every* $i \geq 0$,
2. $|y| > 0$,
3. $|xy| \leq p$.

- Use the fact that regular languages only have finite memory
- An automaton's memory is represented by the states, i.e., if a word is longer than the number of states (=available memory), some state must be repeated twice in the accepting path. $\rightarrow$ cycle!
- Then this accepting path can be divided up into three parts: x (leading to the cycle), y (the cycle), z (path from cycle to accept)

Pumping Lemma - Regular Languages

Lemma (Pumping Lemma)

If A is a regular language, then there is a number p, called the pumping length, where if w is a word in A of length $\geq p$ then w can be divided into three parts, $w = xyz$, such that

1. $xy^iz \in A$ *for every* $i \geq 0$,
2. $|y| > 0$,
3. $|xy| \leq p$.

- Use the fact that regular languages only have finite memory
- An automaton's memory is represented by the states, i.e., if a word is longer than the number of states (=available memory), some state must be repeated twice in the accepting path. $\rightarrow$ cycle!
- Then this accepting path can be divided up into three parts: x (leading to the cycle), y (the cycle), z (path from cycle to accept)

Pumping Lemma - Regular Languages

- useful tool for showing that a language is *nonregular*

Pumping Lemma - Regular Languages

- useful tool for showing that a language is *nonregular*
  Example: $\{a^n b^n \mid n \geq 0\}$

Pumping Lemma - Regular Languages

- useful tool for showing that a language is *nonregular*
  Example: $\{a^n b^n \mid n \geq 0\}$
- NOT useful for showing a language is regular:

$$\{ca^n b^n \mid n \geq 0\} \cup \{c^k w \mid k \neq 1, w \in \Sigma^* \text{ does not start with c}\}$$

Pumping Lemma - Regular Languages

- useful tool for showing that a language is *nonregular*
  Example: $\{a^n b^n \mid n \geq 0\}$
- NOT useful for showing a language is regular:

$$\{ca^n b^n \mid n \geq 0\} \cup \{c^k w \mid k \neq 1, w \in \Sigma^* \text{ does not start with c}\}$$

- a language that is nonregular, yet every word can be pumped according to pumping lemma! $\rightarrow$ sometimes other tools are required (see, e.g., oblig 2)

Context-free languages

- defined *context-free grammars*: essentialy, a set of rules of the form

$$A \to w$$

  where A is a variable and w is a string of variables and terminals

Context-free languages

- defined *context-free grammars*: essentialy, a set of rules of the form

$$A \to w$$

  where A is a variable and w is a string of variables and terminals
- a grammar G generates a word w if starting with the start variable S the word w can be obtained by sequential application of rules in G

Context-free languages

- defined *context-free grammars*: essentialy, a set of rules of the form

$$A \rightarrow w$$

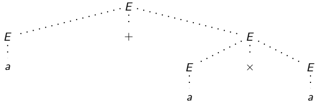
  where A is a variable and w is a string of variables and terminals
- a grammar G generates a word w if starting with the start variable S the word w can be obtained by sequential application of rules in G
- a word w is *ambiguously generated* if there are two or more *leftmost derivations* of w

Context-free languages

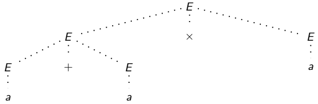
- defined *context-free grammars*: essentialy, a set of rules of the form

$$A \to w$$

  where A is a variable and w is a string of variables and terminals
- a grammar G generates a word w if starting with the start variable S the word w can be obtained by sequential application of rules in G
- a word w is *ambiguously generated* if there are two or more *leftmost derivations* of w



Intuitively corresponds to $a + (a \times a)$



Intuitively corresponds to $(a + a) \times a$

Context-free languages

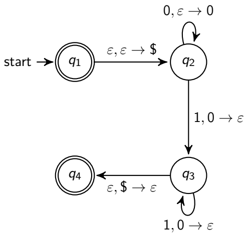
- Context-free languages are accepted by *pushdown automata*: an NFA with an additional stack

Context-free languages

- Context-free languages are accepted by *pushdown automata*: an NFA with an additional stack
- in each transition, we are allowed to pop off and/or push on to the stack.

Context-free languages

- Context-free languages are accepted by *pushdown automata*: an NFA with an additional stack
- in each transition, we are allowed to pop off and/or push on to the stack.



Context-free languages

- converting a CFG to a PDA: use the stack to store intermediate strings of a derivation. The PDA nondeterministically guesses which rule to apply next

Context-free languages

- converting a CFG to a PDA: use the stack to store intermediate strings of a derivation. The PDA nondeterministically guesses which rule to apply next
- converting PDA to CFG: much more involved. General idea: For each pair of states p, q in PDA, add a variable A_{pq} to G that generates all strings that take the PDA from p to q with empty stacks (i.e., stack when arriving at p is equal to the stack when arriving at q). Add certain rules according to transition function δ.

Context-free languages

- converting a CFG to a PDA: use the stack to store intermediate strings of a derivation. The PDA nondeterministically guesses which rule to apply next
- converting PDA to CFG: much more involved. General idea: For each pair of states p, q in PDA, add a variable A_{pq} to G that generates all strings that take the PDA from p to q with empty stacks (i.e., stack when arriving at p is equal to the stack when arriving at q). Add certain rules according to transition function δ.
- So, CFG=PDA

Context-free languages

- converting a CFG to a PDA: use the stack to store intermediate strings of a derivation. The PDA nondeterministically guesses which rule to apply next
- converting PDA to CFG: much more involved. General idea: For each pair of states p, q in PDA, add a variable A_{pq} to G that generates all strings that take the PDA from p to q with empty stacks (i.e., stack when arriving at p is equal to the stack when arriving at q). Add certain rules according to transition function δ.
- So, CFG=PDA
- noteworthy: deterministic PDA (DPDA) is *not* equal to PDA, though we haven't covered this in the lecture

Context-free languages

Every CFG can be rewritten into a grammar in Chomsky normal form:

Definition

A grammar is in *Chomsky Normal Form* if every rule is of the form:

$$A \to BC$$
$$A \to a$$

where a is any terminal, A is any variable, B, C are any variables that are not the start variable. In addition the rule $S \to \varepsilon$ is permitted.

Pumping Lemma - CFL

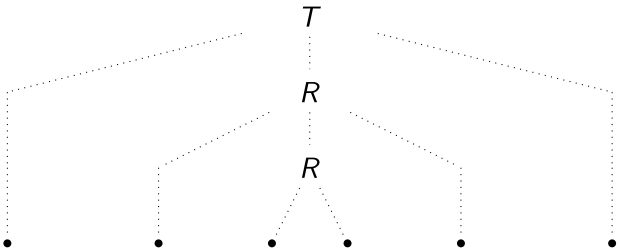
Lemma (Pumping Lemma for CFLs)

For every context-free language A there exists a number p (called the pumping length) where, if s is a word in A of length $\geq p$, then s can be divided into five parts, $s = uvxyz$, satisfying the following conditions:

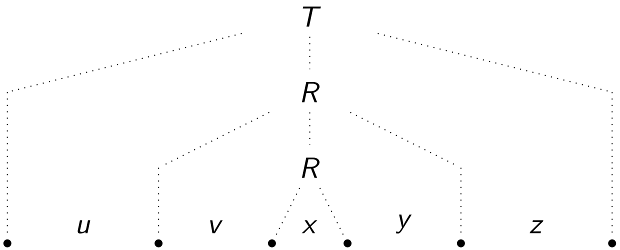
1. $uv^ixy^iz \in A$ for all $i \geq 0$,
2. $|vy| > 0$,
3. $|vxy| \leq p$.

- similar to RL, we exploit the limited memory of CFL's
- If a word is "long enough", the smallest parse tree will contain two occurences of the same variable

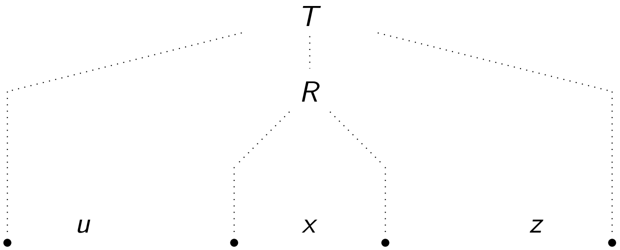
Pumping Lemma - CFLs



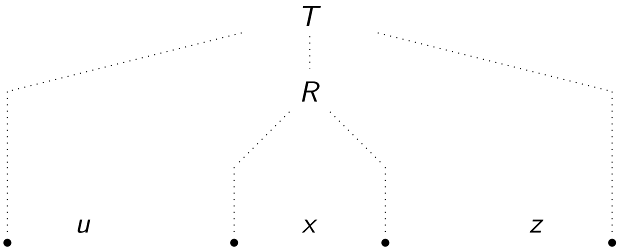
Pumping Lemma - CFLs



Pumping Lemma - CFLs

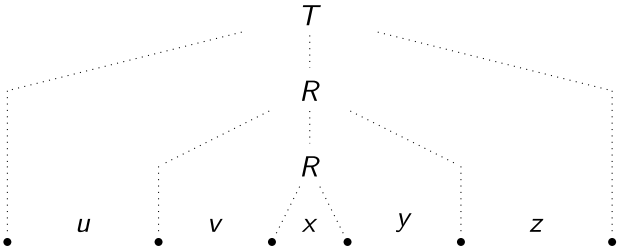


Pumping Lemma - CFLs

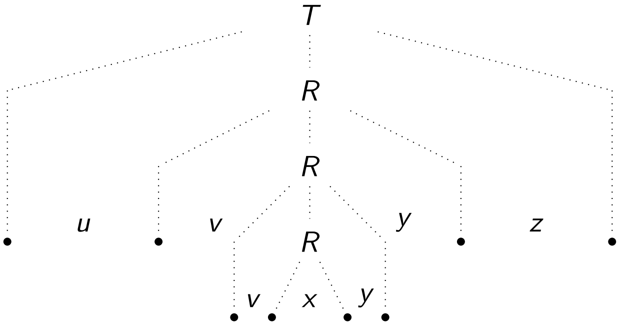


$$\rightarrow uv^0xy^0z = uxz$$

Pumping Lemma - CFLs

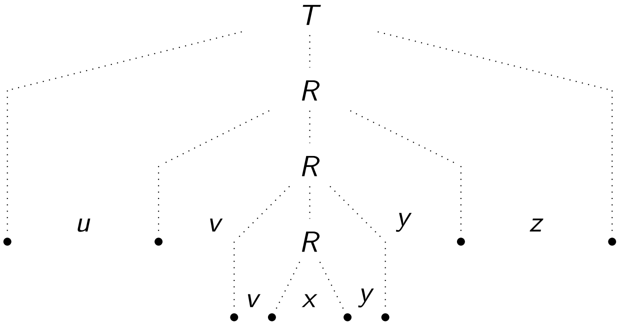


Pumping Lemma - CFLs



- all valid parse trees in G

Pumping Lemma - CFLs



$\rightarrow uv^2xy^2z$, and so on

- all valid parse trees in G

Pumping Lemma - CFLs

- Once again, useful tool for determining if a language is *not* context-free

Pumping Lemma - CFLs

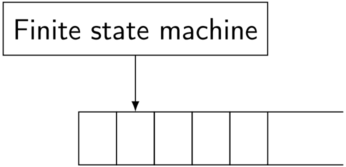
- Once again, useful tool for determining if a language is *not* context-free
- However, just like in the regular case, there exist languages that are not context-free that can be pumped. (analogous to the regular case)

Pumping Lemma - CFLs

- Once again, useful tool for determining if a language is *not* context-free
- However, just like in the regular case, there exist languages that are not context-free that can be pumped. (analogous to the regular case)
- Thus, we have so far seen $\{\text{RL}\} \subsetneq \{\text{CFL}\}$, and that there exist non-context-free languages

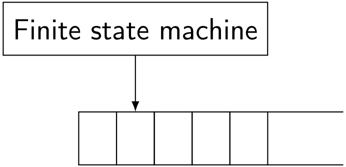
Turing Machines

Defined Turing machines:



Turing Machines

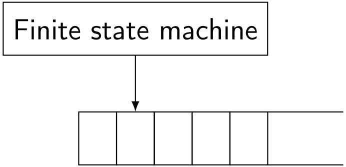
Defined Turing machines:



- a finite state machine with access to an infinite tape

Turing Machines

Defined Turing machines:



- a finite state machine with access to an infinite tape
- modelled by having a read/write head that can move left or right over the tape

Turing Machines

- each of the computational models we had seen so far were special cases of Turing machines

Turing Machines

- each of the computational models we had seen so far were special cases of Turing machines
- different description levels of Turing machiens: high-level ("algorithmic" description, no fine-grained detail on tape operations),

Turing Machines

- each of the computational models we had seen so far were special cases of Turing machines
- different description levels of Turing machiens: high-level (“algorithmic” description, no fine-grained detail on tape operations), low-level (description of how the head operates on tape),

Turing Machines

- each of the computational models we had seen so far were special cases of Turing machines
- different description levels of Turing machiens: high-level (“algorithmic” description, no fine-grained detail on tape operations), low-level (description of how the head operates on tape), implementation level (formal definition of the Turing machine)
- It is important to remember how high-level things can be implemented by tape manipulation, however formal definitions of Turing machines can be cumbersome

Turing Machines

Turing machines are a bit different from the other automata:
DFA/PDA:

- could only read input once (and never move backwards over the input)

Turing Machines

Turing machines are a bit different from the other automata:
DFA/PDA:

- could only read input once (and never move backwards over the input)
- would only accept after having read the entire input (reject if no computational branches accept)

Turing Machines

Turing machines are a bit different from the other automata:
DFA/PDA:

- could only read input once (and never move backwards over the input)
- would only accept after having read the entire input (reject if no computational branches accept)
- either finite memory (DFA), or restricted access to memory (PDA)

Turing Machines

Turing machines are a bit different from the other automata:

DFA/PDA:

- could only read input once (and never move backwards over the input)
- would only accept after having read the entire input (reject if no computational branches accept)
- either finite memory (DFA), or restricted access to memory (PDA)

TM:

- can move left and right across it's tape

Turing Machines

Turing machines are a bit different from the other automata:
DFA/PDA:

- could only read input once (and never move backwards over the input)
- would only accept after having read the entire input (reject if no computational branches accept)
- either finite memory (DFA), or restricted access to memory (PDA)

TM:

- can move left and right across it's tape
- if enters accept/reject state, immediately stops computing

Turing Machines

Turing machines are a bit different from the other automata:

DFA/PDA:

- could only read input once (and never move backwards over the input)
- would only accept after having read the entire input (reject if no computational branches accept)
- either finite memory (DFA), or restricted access to memory (PDA)

TM:

- can move left and right across it's tape
- if enters accept/reject state, immediately stops computing
- unrestricted access to infinite memory

Turing Machines

A language accepted by a Turing machine is called Turing-recognizable. If the machine halts on every input, then the language it recognizes is called decidable.

Turing Machines

Have looked at Turing machine variants, seen that they are equivalent:

- the LRS Turing machine (the head can move left, right, or stay put)
- the multitape Turing machine (multiple tapes, multiple heads)
- the nondeterministic Turing machine
- the enumerator
- NFA with two stacks
- ...

All computational models with unlimited access to infinite memory that can perform finite work in one step are equivalent to a Turing machine!

Church-Turing Thesis

- Church and Turing independently formalized the notion of algorithm

Church-Turing Thesis

- Church and Turing independently formalized the notion of algorithm
- Previous, intuitive notion: a method according to which after a finite number of operations an answer is given (paraphrased, many formulations)

Church-Turing Thesis

- Church and Turing independently formalized the notion of algorithm
- Previous, intuitive notion: a method according to which after a finite number of operations an answer is given (paraphrased, many formulations)
- Formal: an algorithm is a decidable Turing machine (deciders)

Church-Turing Thesis

- Church and Turing independently formalized the notion of algorithm
- Previous, intuitive notion: a method according to which after a finite number of operations an answer is given (paraphrased, many formulations)
- Formal: an algorithm is a decidable Turing machine (deciders)
- Church Turing thesis: each intuitive definition of algorithms can be described by decidable Turing machines

Decidability

Considered acceptance, emptiness, and equivalence problems for computational models, e.g.:

$$A_{TM} = \{\langle M, w\rangle \mid M \text{ is a Turing machine that accepts } w\}$$

Decidability

Considered acceptance, emptiness, and equivalence problems for computational models, e.g.:

$$A_{TM} = \{\langle M, w\rangle \mid M \text{ is a Turing machine that accepts } w\}$$

We showed various decidability/undecidability results for languages:

	$x \in L$	$L = \emptyset$	$L = \Sigma^*$	$L = K$
regular	✓	✓	✓	✓
CFL	✓	✓	X	X
LBA	✓	X	X	X
decidable	✓	X	X	X
Turing-rec.	X	X	X	X

Undecidability

Considered the halting problem:

$$HALT_{TM} = \{\langle M, w \rangle \mid M \text{ is a TM that halts on input } w\}$$

Undecidability

Considered the halting problem:

$$HALT_{TM} = \{\langle M, w\rangle \mid M \text{ is a TM that halts on input } w\}$$

- $HALT_{TM}$ is undecidable.

Undecidability

Considered the halting problem:

$$HALT_{TM} = \{\langle M, w \rangle \mid M \text{ is a TM that halts on input } w\}$$

- $HALT_{TM}$ is undecidable.
- Thus, it is *algorithmically unsolvable* to determine whether a given problem will terminate!

Undecidability

Considered the halting problem:

$$HALT_{TM} = \{\langle M, w\rangle \mid M \text{ is a TM that halts on input } w\}$$

- $HALT_{TM}$ is undecidable.
- Thus, it is *algorithmically unsolvable* to determine whether a given problem will terminate!
- saw PCP yesterday: given a set of dominoes, does there exist a match?

Undecidability

Considered the halting problem:

$$HALT_{TM} = \{\langle M, w\rangle \mid M \text{ is a TM that halts on input } w\}$$

- $HALT_{TM}$ is undecidable.
- Thus, it is *algorithmically unsolvable* to determine whether a given problem will terminate!
- saw PCP yesterday: given a set of dominoes, does there exist a match?
- also undecidable.

Undecidability

Considered the halting problem:

$$HALT_{TM} = \{\langle M, w \rangle \mid M \text{ is a TM that halts on input } w\}$$

- $HALT_{TM}$ is undecidable.
- Thus, it is *algorithmically unsolvable* to determine whether a given problem will terminate!
- saw PCP yesterday: given a set of dominoes, does there exist a match?
- also undecidable.

→ decidability relates to more things than just Turing machines!

Wrap-up

Connecting Chomsky and Turing, the Chomsky hierarchy:

Wrap-up

Connecting Chomsky and Turing, the Chomsky hierarchy:

- Type-0: recursively enumerable, i.e., Turing-recognizable languages.
- Type-1: context-sensitive languages.
- Type-2: context-free languages.
- Type-3: regular languages.

Wrap-up

Connecting Chomsky and Turing, the Chomsky hierarchy:

- Type-0: recursively enumerable, i.e., Turing-recognizable languages.
- Type-1: context-sensitive languages.
- Type-2: context-free languages.
- Type-3: regular languages.

We haven't gone through Type-1 (extra lecture at the end of the semester, if desired), however we have seen the computational model that accepts them: linear bounded automata (LBA) and seen that these are decidable.

What's next?

Complexity!

- not so much about decidability vs. undecidability...most of what we'll consider will be decidable, i.e., algorithmically solvable.

What's next?

Complexity!

- not so much about decidability vs. undecidability...most of what we'll consider will be decidable, i.e., algorithmically solvable.
- ...but how hard are these problems? How can they be compared with one another

What's next?

Complexity!

- not so much about decidability vs. undecidability...most of what we'll consider will be decidable, i.e., algorithmically solvable.
- ...but how hard are these problems? How can they be compared with one another
- related to reducibility, computable functions

What's next?

Complexity!

- not so much about decidability vs. undecidability...most of what we'll consider will be decidable, i.e., algorithmically solvable.
- ...but how hard are these problems? How can they be compared with one another
- related to reducibility, computable functions
- *highly* relevant for anything within computer science, be it crypto/security, programming, theoretical work (AI, databases)