Automata Theory and Formal Languages

Introduction to Automata Theory

- **What is Automata Theory?**
- **Central Concepts of Automata Theory**
- **Formal Proofs**

What is Automata Theory?

Automata Theory

- **Automata theory** is the study of abstract ***computing devices (machines)***.

- In 1930s, **Turing** studied an abstract machine (***Turing machine***) that had all the capabilities of today's computers.
  - Turing's goal was to describe precisely the boundary between what ***a computing machine could do and what it could not do.***

- In 1940s and 1950s, simpler kinds of machines (**finite automata**) were studied.
  - **Chomsky** began the study of **formal grammars** that have close relationships to abstract automata and serve today as the basis of some important software components.

Why Study Automata?

- **Automata theory** is the ***core of computer science***.

- Automata theory presents ***many useful models for software and hardware***.
  - In compilers we use finite automata for lexical analyzers, and push down automatons for parsers.
  - In search engines, we use finite automata to determine tokens in web pages.
  - Finite automata model protocols, electronic circuits.
  - Context-free grammars are used to describe the syntax of essentially every programming language.
  - Automata theory offers many useful models for natural language processing.

- When developing solutions to real problems, we often confront the ***limitations of what software can do.***
  - **Undecidable** things – ***no program whatever can do it***.
  - **Intractable** things – ***there are programs, but no fast programs.***

Automata, Computability and Complexity

- **Automata**, **Computability** and **Complexity** are linked by the question:
  - ***"What are the fundamental capabilities and limitations of computers?"***
- In **complexity theory**, the objective is to classify problems *as* ***easy problems*** and ***hard problems***.
- In **computability theory**, the objective is to classify problems as **solvable problems** and non-solvable problems.
  - Computability theory introduces several of the concepts used in complexity theory.
- **Automata theory** deals with the definitions and properties of mathematical models of computation.
  - Finite automata are used in text processing, compilers, and hardware design.
  - Context–free grammars are used in programming languages and artificial intelligence.
  - Turing machines represent computable functions.

Central Concepts of Automata Theory

Central Concepts of Automata Theory - Alphabets

- An **alphabet** is a finite, non empty set of symbols.
- We use the symbol $\sum$ for an alphabet.

- $\sum = \{0,1\}$ - binary alphabet
- $\sum = \{a,b,c,\ldots,z\}$ - lowercase letters
- The set of ASCII characters is an alphabet.

Central Concepts of Automata Theory - Strings

- A **string** is a sequence of symbols chosen from some alphabet.
- A string sometimes is called as **word**.

- 01101 is a string from the alphabet $\Sigma = \{0,1\}$.
  - Some other strings: 11, 010, 1, 0

- The **empty string**, denoted as **ε**, is a string of zero occurrences of symbols.

- **Length of string:** number of symbols in the string
  - $|ab| = 2$ $|b| = 1$ $|\boldsymbol{\varepsilon}| = 0$

Central Concepts of Automata Theory - Strings

Powers of an alphabet:

- If Σ is an alphabet, the set of all strings of a certain length from the alphabet by using an exponential notation.
- Σ^k is the set of strings of length k from Σ.
- Let $\Sigma = \{0,1\}$. $\Sigma^0 = \{\varepsilon\}$ $\Sigma^1 = \{0,1\}$ $\Sigma^2 = \{00,01,10,11\}$
- The set of all strings over an alphabet is denoted by Σ^*.

$$\Sigma^* = \Sigma^0 \cup \Sigma^1 \cup \Sigma^2 \cup \ldots$$

$$\Sigma^+ = \Sigma^1 \cup \Sigma^2 \cup \ldots$$ - set of nonempty strings

Concatenation of strings

- If **x** and **y** are strings **xy** represents their concatenations.
- If **x** = abc and **y** = de then **xy** = abcde

Central Concepts of Automata Theory – (Formal) Languages

- A set of strings that are chosen from $\sum^*$ is called as a **language**.
- If $\sum$ is an alphabet, and $\mathbf{L \subseteq \sum^*}$, then L is a **language** over $\sum$.

- A language over $\sum$ may not include strings with all symbols of $\sum$.
- Some Languages:
  - The language of all strings consisting of n 0's followed by n 1' for some $n \geq 0$: {ε, 01, 0011, 000111, …}
  - $\sum^*$ is a language
  - Empty set is a language. The empty language is denoted by Φ.
  - The set {ε} is a language, {ε} is not equal to the empty language.
  - The set of all identifiers in a programming language is a language.
  - The set of all syntactically correct C programs is a language.
  - Turkish, English are languages.

Set-Formers to Define Languages

- A **set-former** is a common way to define a language

  Set-former: {w | something about w}

  {w | w consists of equal number of 0's and 1's}

  {w | w is a binary integer that is prime}

Sometimes we replace w with an expression

$\{0^n 1^n \mid n \geq 1\}$

$\{0^i 1^j \mid 0 \leq i \leq j\}$

Language – Decision Problem

- In automata theory, a **decision problem** is the question of deciding whether a given string is a member of a particular language.
- **If $\sum$ is an alphabet, and L is a language over $\sum$, then the decision problem is:**
  Given a string w in $\sum^*$, decide whether or not w is in L.
- In order to make decision requires some computational resources.
  - Deciding whether a given string is a correct C identifier
  - Deciding whether a given string is a syntactically correct C program.
- Some decision problems are simple, some others are harder.
- A decision question may ***require exponential resources in the size of its input.***
- A decision question may be ***unsolvable***.

Automata

- **Automata** (singular **Automaton**) are abstract mathematical devices that can
  - Determine membership in a language (set of strings)
  - Transduce strings from one set to another
- They have all the aspects of a computer
  - input and output
  - memory
  - ability to make decisions
  - transform input to output
- Memory is crucial:
  - Finite Memory
  - Infinite Memory

Automata

- We have different types of automata for different classes of languages.
  - **Finite State Automata** (for ***regular languages***)
  - **Pushdown Automata** (for ***context-free languages***)
  - **Turing Machines** (for ***Turing recognizable languages - recursively enumerable languages***)
    - Decision problem for Turing recognizable languages are solvable.
    - There are languages that are not Turing recognizable, and the decision problem for them is unsolvable.

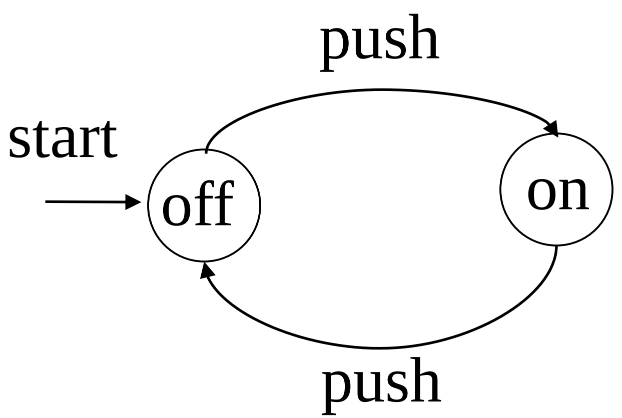
- Automata differ in
  - the amount of memory then have (finite vs infinite)
  - what kind of access to the memory they allow.

- Automata can behave **deterministically** or **non-deterministically**
  - For a **deterministic automaton**, there is only one possible alternative at any point, and it can only pick that one and proceed.
  - A **non-deterministic automaton** can at any point, among possible next steps, pick one step and proceed.

Finite Automata

- **Finite automata** are ***finite collections of states with transition rules*** that take you from one state to another.
- A **finite automaton** has **finite number of states**.
- The ***purpose of a state*** is to remember the relevant portion of the history.
  - Since there are only a ***finite number of states***, the entire history cannot be remembered.
    - So the system must be designed carefully to remember what is important and forget what is not.
  - The advantage of having only a finite number of states is that we can implement the system with a fixed set of resources.

A Simple Finite Automaton – On/Off Switch



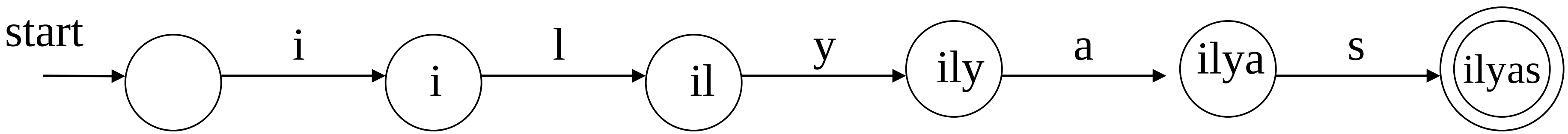
In a **finite automaton**:

- **States** are represented by **circles**.
- **Accepting (final) states** are represented by **double circles**.
- One of the states is a **starting state**.
- **Arcs** represent **state transitions** and **labels on arcs** represent **inputs** (external influences) causing transitions.

- The on/off switch remembers whether it is in the on-state or the off-state.
  - It allows the user to press a button whose effect is different depending on the state of the switch.

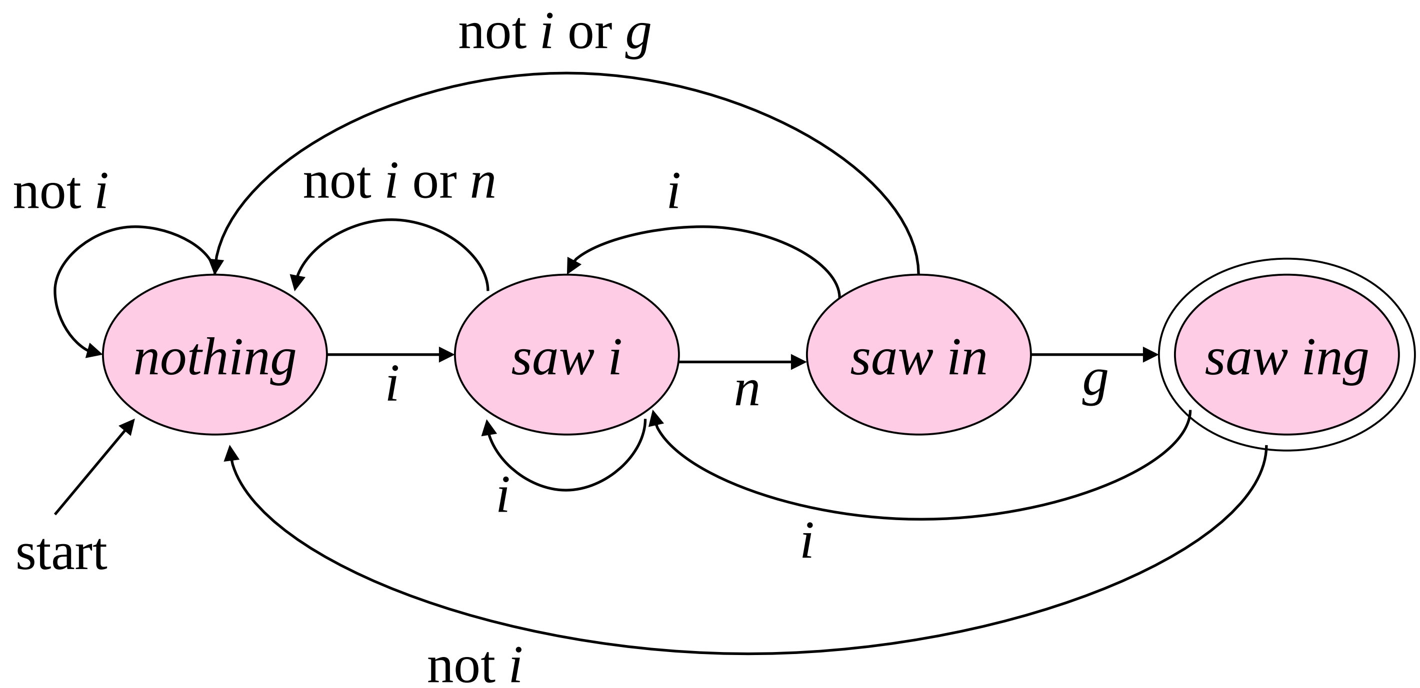
A Simple Finite Automaton – Recognizing A Word

- A simple finite automaton to recognize the string "`ilyas`"



- The language of this finite state automaton is {ilyas}

A Simple Finite Automaton – Recognizing Strings Ending in "ing"



- The language of this automaton is the set of all strings ending in "ing".
  - i.e. {ing, aing, bing, going, coming, inging, …}

Formal Proofs

Formal Proofs

- When we study automata theory, we encounter theorems that we have to prove.
- There are different forms of proofs:
  - Deductive Proofs
  - Inductive Proofs
  - Proof by Contradiction
  - Proof by a counter example (disproof)
- To create a proof may NOT be so easy.

Deductive Proofs

- A **deductive proof** consists of a sequence of statement whose truth leads us from some *initial statement* (hypothesis or given statements) to a *conclusion statement*.

- Each step of a deductive proof MUST follow from a given fact or previous statements (or their combinations) by an accepted **logical principle (inference rules)**.

  - A logical principles guarantees that if its **premises** are correct(true), its **conclusion** is correct (true) too.

premise$_1$ … premise$_n$		**Hypothesis**
-------------------------------	**Logical Principle**	
conclusion		**Conclusion**

- The theorem that is proved when we go from a hypothesis H to a conclusion C is the statement **’’if H then C’’.** We say that C is deduced from H.

Deductive Proofs

Example: Proof of a Theorem

- Assume that the following theorem (initial statement) is given:
  - Given Theorem. (initial statement): **If $x \geq 4$, then $2^x \geq x^2$**
  - We are not going to prove this theorem, we assume that it is true.
    - If we want we can prove this theorem using proof by induction.

- **Theorem to be proved**:

If x is the sum of the squares of four positive integers, then $2^x \geq x^2$

Hypothesis (x is the sum of the squares of four positive integers)

Conclusion ($2^x \geq x^2$)

Deductive Proofs

Example: Proof of a Theorem

Proof of

If x is the sum of the squares of four positive integers, then $2^x \geq x^2$

Statement	Justification
1. If $x \geq 4$, then $2^x \geq x^2$	Given theorem
2. $x = a^2 + b^2 + c^2 + d^2$	Given
3. $a \geq 1$ $b \geq 1$ $c \geq 1$ $d \geq 1$	Given
4. $a^2 \geq 1$ $b^2 \geq 1$ $c^2 \geq 1$ $d^2 \geq 1$	From (3) and principle of arithmetic
5. $x \geq 4$	From (2), (4) and principle of arithmetic
6. $2^x \geq x^2$	From (1) and (5)

If-And-Only-If Statements

- Some times theorems contain **if-and-only-if** statements.
  - A if and only if B
  - A iff B
  - A is equivalent to B
- In this case we have to prove in both directions. In order to prove **A if and only if B**, we have to prove the following two statements:

  1. **If-Part:** **if B then A**
  2. **Only-If-Part:** **if A then B**

A Sample iff Theorem:

Let x be a real number. Then $\lfloor x \rfloor = \lceil x \rceil$ if and only if x is an integer.

Remember: $\lfloor x \rfloor$ is the *floor* of real number x is the greatest integer equal to or less than x

$\lceil x \rceil$ is the *ceiling* of real number x is the least integer equal to or greater than x

Proof of an iff Theorem

Let x be a real number. Then $\lfloor x \rfloor = \lceil x \rceil$ if and only if x is an integer.

If-Part:

- Given that x is an integer.
- By definitions of ceiling and floor operations. $\lfloor x \rfloor = x$ and $\lceil x \rceil = x$
- Thus, $\lfloor x \rfloor = \lceil x \rceil$.

Only-If-Part:

- Given that $\lfloor x \rfloor = \lceil x \rceil$
- By definitions of ceiling and floor operations. $\lfloor x \rfloor \leq x$ and $\lceil x \rceil \geq x$
- Since given that $\lfloor x \rfloor = \lceil x \rceil$, $\lceil x \rceil \leq x$ and $\lceil x \rceil \geq x$
- By the properties of arithmetic inequalities, $\lceil x \rceil = x$
- Since $\lceil x \rceil$ is always an integer, x MUST be integer too. □

Inductive Proofs

- An **inductive proof** has three parts:
  - Basis
  - Inductive Hypothesis
  - Inductive Step (induction)

- Basis can be one case or more than one case.

Inductive Proofs -- Theorem: $\sum_{i=1}^{n} i = \frac{n(n+1)}{2}$ for all n≥1

Proof : (by induction on n)

Basis: n = 1 $\sum_{i=1}^{1} i = \frac{1(1+1)}{2}$ 1=1

Inductive Hypothesis: Suppose that $\sum_{i=1}^{k} i = \frac{k(k+1)}{2}$ for some k ≥ 1.

Inductive Step (Induction): We have to show that $\sum_{i=1}^{k+1} i = \frac{(k+1)(k+2)}{2}$

$$\sum_{i=1}^{k+1} i = \sum_{i=1}^{k} i + (k+1)$$

$$= \frac{k(k+1)}{2} + (k+1) \quad \text{by the inductive hypothesis}$$

$$= \frac{k(k+1) + 2(k+1)}{2} = \frac{(k+1)(k+2)}{2}$$

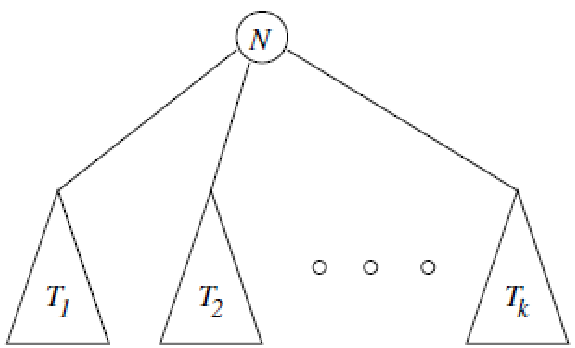
It follows that $\sum_{i=1}^{n} i = \frac{n(n+1)}{2}$ for all n ≥ 1. □

Structural Inductions

- We need to prove statements about ***recursively defined structures.***
- Like *inductions* all **recursive definitions** have
  - A basis case: one or more elementary structures are defined
  - An inductive step: complex structures are defined in terms of previously defined structures.

A recursive definition of a non-empty tree:

- A single node is a non-empty tree and that node is the root of that tree.
- If $T_1, T_2, \ldots, T_k$ are non-empty trees ($k \geq 1$) and N is a new node, the a new non-empty tree T can be created using new node N, new k edges and $T_1, T_2, \ldots, T_k$ as follows:



where N is the root of the tree

Structural Inductions

- Let |V| be the number nodes and |E| be the number of edges of a non-empty tree T.

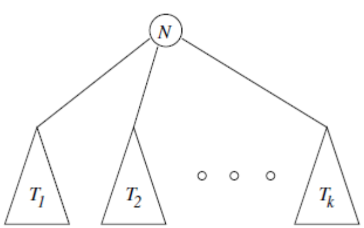
Theorem: **For a non-empty tree T, $|V| = |E| + 1$.**

Proof: **Structural induction on number of nodes.**

Basis: $|V|=1$ The tree contains only one node and no edges ($|E|=0$). Thus $1=0+1$.

Inductive Hypothesis: Suppose that for a non-empty tree T with m nodes where $1 \leq m \leq n$, $|V|=|E|+1$

Induction: Let T be a non-empty tree with n+1 nodes. T must be created as follows:



Each of trees $T_1,\ldots,T_k$ must contain nodes less than or equal to n.

So, we can apply IH to each of trees $T_1,\ldots,T_k$. Thus, $|V_1|=|E_1|+1 \;\ldots\; |V_k|=|E_k|+1$

For T, $|V| = |V_1|+\ldots+ |V_k|+1$ $\qquad |E| = |E_1|+\ldots+ |E_k|+k$

$|V| = |V_1|+\ldots+ |V_k|+1 = |E_1|+1+\ldots+ |E_k|+1+1$ by IH

$= |E_1|+\ldots+ |E_k|+k+1 = |E| + 1$ $\square$

Proving Equivalences about Sets

- In order to prove two sets are equal (S = T), we have to prove that
  1. If x is a member of S, then x is also a member of T   ($S \subseteq T$), and
  2. If x is a member of T, than x is also a member of S ($T \subseteq S$),

Theorem: $R \cup (S \cap T) = (R \cup S) \cap (R \cup T)$

We have to show that

1. If x is in $R \cup (S \cap T)$, than x is in $(R \cup S) \cap (R \cup T)$, and
2. If x is in $(R \cup S) \cap (R \cup T)$, than x is in $R \cup (S \cap T)$

Proof of $R \cup (S \cap T) = (R \cup S) \cap (R \cup T)$

Proof of If-Part:

	Statement	Justification
1.	x is in $R \cup (S \cap T)$	Given
2.	x is in R or x is in $(S \cap T)$	(1) and definition union
3.	x is in R or x is in both S and T	(2) and definition of intersection
4.	x is in $(R \cup S)$	(3) and definition of union
5.	x is in $(R \cup T)$	(3) and definition of union
6.	x is in $(R \cup S) \cap (R \cup T)$	(4), (5) and definition of intersection

Proof of $R \cup (S \cap T) = (R \cup S) \cap (R \cup T)$

- **Proof of Only-If-Part:**

	Statement	Justification
1.	x is in $(R \cup S) \cap (R \cup T)$	Given
2.	x is in $(R \cup S)$	(1) and definition intersection
3.	x is in $(R \cup T)$	(1) and definition of intersection
4.	x is in R or x is in both S and T	(2), (3) and reasoning about unions
5.	x is in R or x is in $(S \cap T)$	(4) and definition of intersection
6.	x is in $R \cup (S \cap T)$	(5) and definition of union

Proof by Contradiction

- Another way to prove a statement of the form "**if H then C**" is to prove the statement.

  "**H and not C implies falsehood**"

- In order create the proof:
  - Start by assuming both the **hypothesis H** and the **negation of the conclusion C**.
  - Complete the proof by showing that something known to be **false** follows logically from **H** and **not C**
  - Then, conclude **C**

- This form of proof is called **proof by contradiction**.