Theoretical Computer Science – Main Points

Ahmed Nouralla

Lecture 1:

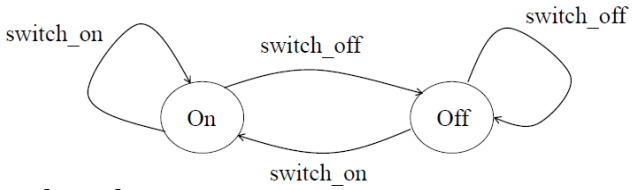
- **Computability** is about what can be computed.
- **Complexity** is about how efficiently can it be computed
- **Finite State Automata (FSA):** no temporary memory
  - **Example:** Elevators, Vending Machines ("small" computing power).
- **Pushdown Automata (PDA):** stack (destructive memory)
  - **Example:** Compilers for Programming Languages ("medium" computing power).
- **Turing Machines (TMs):** random (non-sequential) access memory
  - **Any Algorithm** ("highest" known computing power).

Lecture 2:

- **Mathematical abstraction: allows to focus on the important aspects of a problem**
- **Language** is a set of sentences, each finite in length, and constructed out of a finite set (an alphabet). $L \in A^*$
- **Kleene star**: a unary operator that can applied on a given set of characters (Alphabet)
  - It produces an infinite set of all possible strings that can be constructed by concatenating elements from this set into one string.
  - It has the empty string present.
- **Exponential notation on an alphabet:** $\Sigma^k = \{x \in \Sigma^* \mid |x| = k\}$

Lecture 3:

- **Mathematical description of the light switch system as an example of FSA**
  - FSA = <Q, A, δ, q0, F>
  - Q = {“On”, “Off”}
  - A = {“switch_off”, “switch_on”}
  - q0 = “Off”
  - F = {“Off”}



  - δ = (Q × A, R) // function is a set, with a relation on it.
  - R = { ({“On”, “switch_off” }, "Off"), ({"Off", “switch_on”}, "On"), ({"On", “switch_on” }, "On"), ({"Off", “switch_off” }, "Off") } // Relation is a set of pairs.
  - δ* is a function that tells you a final state given an initial state and a string (moving sequence).
  - The string is a concatenation of several elements from the alphabet, given in a specific order. In other words, the moving sequence is an element of A*
  - Note that A* is the set of all possible strings that can be constructed from an alphabet A
  - δ* is considered "valid" iff it returns a state that belongs to the final states set. In this case, we say that the moving sequence is accepted.
- **Finite State Transducer:**
  - An FSA that works on two memory tapes, input tape and output tape
  - It reads input from the tape sequentially, does some translation, and produce characters on the output tape.
  - The ordinary FSA is considered to have only one tape.
  - FSA defines a formal language by defining a set of accepted strings
  - FST defines relations between sets of strings.
  - FST can output to a tape along with making a transition.
  - FST = <Q, A, δ, q0, F, O, η>
    - O: output alphabet
    - η: Q × I → O* Translation function.

Common question: build an FSA that reads a binary string and accepts if its decimal equivalent is divisible by k.

Solution:

- q0 will be the initial and the only final state.
- i in qi will represent the reminder when dividing the input number by k.
- We construct the right table if k is odd, and left one if it’s even:
- We use the table to draw the diagram.

State	0	1
q0	q0	q1
q1	q2	q3
q2	q4	q5
…	…	…
…	…	q(k-1)
…	q0	q1
…	…	…
q(k-1)	…	q(k-1)

State	0	1
q0	q0	q1
q1	q2	q3
q2	q4	q5
…	…	…
…	q(k-1)	q0
…	q1	q2
…	…	…
q(k-1)	q(k-1)	q0

Lecture 4:

- **Peano axioms:**
  - 0 is a natural number.
  - Equality relation is Reflexive, Symmetric, and Transitive.
  - Natural numbers are closed under equality relation.
  - Successor function S(n) is injective. Natural numbers are closed under it.
  - No natural number whose successor is zero.
  - Axiom of induction: **If**
    - φ is a predicate (Function from some set to the set B = {true, false}.
    - φ(0) is true.
    - For every natural number n, φ(n) being true implies that φ(S(n)) is true.
    - **Then** φ(n) is true for every natural number.
- **A set is closed** w.r.t. an operation **if** the operation is applied to elements of the set and the result is still an element of the set.
- **Regular language:** Language recognized by a FSA.
  - Regular language is closed w.r.t. set theoretic operations (Union, Intersection, etc.) + concatenation, and Kleene star.
- **A string belongs to a Language** if the moving sequence obtained from applying it to the FSA that recognizes that language while it's is in its initial state, it leads to a final state.
- **If a language is not empty**, then there exists a path in FSA from the initial state to the final state, it's empty otherwise.
- **The language is infinite** if there is a cycle in the FSA diagram.

Operations on FSAs

- **Intersection** of two FSAs represents the parallel run of them.
  - A double-state node is final iff **both** of its states are final.
  - **Formally:**
    - $A_1 = < Q_1, I, \delta_1, q_0^1, F_1 >$ $A_2 = < Q_2, I, \delta_2, q_0^2, F_2 >$
    - $A_1 \cap A_2 = < Q_1 \times Q_2, I, \delta, <q_0^1, q_0^2>, \mathbf{F_1 \times F_2} >$
    - δ (<q_1, q_2>, i) = < δ_1 (q_1, i), δ_2(q_2, i) >
  - The resulting FSA accepts a string iff both A_1 and A_2 accepts it at the same time.
- **Union** of two FSAs represents a new FSA that accepts strings that are accepted either by the first one or by the second one, or both of them.
  - A double-state node is final iff **at least one** of its states is final.
  - **Formally:**
    - $A_1 = < Q_1, I, \delta_1, q_0^1, F_1 >$ $A_2 = < Q_2, I, \delta_2, q_0^2, F_2 >$
    - $A_1 \cup A_2 = < Q_1 \times Q_2, I, \delta, <q_0^1, q_0^2>, \mathbf{F_1 \times Q_2 \cup Q_1 \times F_2}>$
    - δ (<q_1, q_2>, i) = < δ_1 (q_1, i), δ_2(q_2, i) >
- **Complement** of an FSA represents an FSA that accepts only strings that are rejected by the original one, and rejects otherwise.
  - We obtain it by swapping final and non-final states.
  - It's necessary to complete the FSA before doing so.
  - **Formally:** $F^C = Q - F$
- **Difference** between two FSAs A and B represents an FSA that accepts only strings that are accepted by A and not by B.
  - A double-state node <a, b> is final iff **'a' is final** in the **first** FSA and **'b'** is **not**.

- **Formally:**
  - $A_1 = < Q_1, I, \delta_1, q_0^1, F_1 >$ $A_2 = < Q_2, I, \delta_2, q_0^2, F_2 >$
  - $A_1 \cup A_2 = < Q_1 \times Q_2, I, \delta, <q_0^1, q_0^2>, \mathbf{F_1 - F_2}>$
  - $\delta (<q_1, q_2>, i) = < \delta_1 (q_1, i), \delta_2(q_2, i) >$

- The result of any operation on two FSAs is an FSA that is represented in diagrams using double-state nodes (Nodes that contain a pair of states instead of one) of every possible combination.
- Total number of nodes in the resulting FSA = n*m where n is the number of nodes in the first FSA and m is the number of nodes in the second one.
- There is only one double-state node that is initial. The one that contains the two initial states.
- **De Morgan laws apply on FSAs** $\overline{A \cup B} = \overline{A} \cap \overline{B},$ $\overline{A \cap B} = \overline{A} \cup \overline{B},$

- **Comparison between FSA and TM:**

Turing machine	**Finite State Automata**
More expressive (More programs can be expressed through it).	Less expressive.
Allows non-terminating programs.	Has a finite number of states. Can run infinitely given an infinite string.
Accepts a string if it halts and says yes.	Accepts a string if it leads to one of the final states.
If it doesn't halt, we can't determine.	If it didn't reach a final state, string is rejected.

- **Syntax:** is the set of rules, principles, and processes that govern the structure of sentences in a given language.
- **Semantics:** is the linguistic (the study of meaning in language).
  - It focuses on the relationship between signifiers (words, phrases), symbols and what they stand for (denotation: translation of a sign to its meaning).

Lecture 5:

- **Regular language: A language recognized by a FSA (Or any fixed-Memory automaton).**
    - It can be represented using FSA/NFSA/RegEx/Regular Grammars.
    - It's closed with respect to set operations, concatenation and Kleene star.
- **Pumping lemma.**
    - All sufficiently long words in a regular language may be pumped. That is, have a middle section of the word repeated an arbitrary number of times, to produce a new word that also lies within the same language.
    - For any string x that belongs to some language L
        - If |x| >= |Q| then the string can pumped.
    - Pumping Lemma is used to proof that a language is not regular.
- **If we know that:**
    - L is an infinite regular language (We can construct an FSA that can recognize it).
- **Then pumping lemma guarantees that:**
    - **There for any string x that**
        - Belongs to the language.
        - Has a length bigger than or equal the critical length (The number of states in the FSA that can recognize L)
    - x can be expressed as a concatenation of a three strings y, w, z such that if the string yw^iz belongs to L for any integer i.
    - w cannot be the empty string, but y and z can be.
    - **Formally:**
        - $\forall x \in L$
        - $|x| \geq m, m = p = |Q|$
        - $x = ywz$
        - $yw^iz \in L, i \in N$
        - $|w| \geq 1$
        - $|yw| \leq m$
- **PushDown Automata (PDA): FSA with a theoretically infinite stack.**
    - It can use the top of stack to decide the transition to make.
    - It can do operations on the stack while transiting from a state to another.
    - The Stack has its own alphabet "The set of possible elements that can be pushed into the stack".
    - A single move of a PDA will depend on:
        - The current state.
        - The next input (it could be no symbol: **ε**)
        - The symbol currently on top of the stack.
    - PDA will be assumed to begin operation with an initial start symbol Z_0 on the top of its stack, Z_0 is not essential, but useful to simplify definitions
    - Z_0 is on top means that the stack is effectively empty.
    - **Formally:** PDA = < Q, I, Γ, δ, q_0, Z_0, F >
        - δ: $Q \times (I \cup \varepsilon) \times \Gamma \rightarrow Q \times \Gamma^*$
        - PDA **Configuration** is a triple (q, x, γ) represents the current "condition" of the PDA, q is the state we are currently at, x is the string unread, γ is the string of symbols currently in the stack.

Lecture 6:

- **Acceptance:**
  - **By final state:** Input is consumed and PDA is in a final state.
  - **By empty stack:** Input is consumed and stack is empty.
  - Both are the same in NPDA, but they are different in DPDA.
    - In DPDA, if two inputs belong to the same language and one of them is a prefix of the other one, then we can't construct a "empty-stack-acceptance-PDA" that accepts this language.
- **Memory model:** The choice of memory model influences the expressiveness of the computational model; It determines what can be computed and what cannot.
  - **Fixed memory:** A memory that has a fixed size that cannot be changed **(FSA states).**
  - **Finite memory:** A memory that can't be infinite, can be persistent or destructive.
  - **Destructive memory:** Once a symbol is read, it can't be read again. **(PDA stack)**
  - **Persistent memory:** A memory that can be read multiple times **(TM tapes - Von Neumann Architecture)**
- FSAs are used in compilers for lexical analysis which breaks the source code text into small pieces.
- NPDAs are used in compilers for analyzing syntax, and parsing expressions.
- **Spontaneous move:** when the PDA (changes state/manipulates stack) depending on the stack top element only without reading the input.
  - From some configuration, if we have a spontaneous move but we also have an ordinary move at the same time (The transition function from the configuration is defined for the empty string and also some other string), the machine is nondeterministic.
- c1 |-- c2, means there is a transition from configuration c1 to configuration c2
- c1 |-*- c2 means there is a reflexive transitive relation |-- between c1 and c2.
  - |-*- is the sequence of 0 or more moves taking the PDA from config. c1 to c2.
- **Acceptance in PDA.**

$$\forall x \in I^* \ (x \in L \Leftrightarrow c_0=<q_0,x,Z_0> \ |-^*- \ c_F=<q,\varepsilon,\gamma> \text{ and } q \in F)$$

  - For all strings x that belongs to the Language constructed from the alphabet I
  - x belongs to the language iff there is a reflexive transitive relation from the configuration c0 to the configuration cf
  - c0 is the initial configuration of the machine when we are in the initial state and we haven't read the whole x string and the stack is currently effectively empty
  - cf: is the final configuration when we are at one of the final states, we have nothing more to read and the stack currently have any string it doesn't matter.
- **Example of languages:**
  - Recognized by FSA: Any finite language.
  - Recognized by PDA: A^nB^n, WCW^R
  - Recognized by TM: $A^nB^nC^n$, $A^nB^n \cup A^nB^{2n}$
- **Limitations of PDA:**
  - Can be proved using Bar-Hillel lemma.
  - The stack is destructive, once we pop an element, we can't get it back.
  - We need a persistent memory (Memory tapes – Turing Machines).

Lecture 7:

- **Kleene plus:** $L^* = \{\varepsilon\} \cup L^+$
- **Context-Free Languages (CFLs):**
  - A language generated by a context-free grammar.
  - We can always construct a PDA that accepts it.
  - Set of all languages accepted by PDA = Set of all CFLs
- **PushDown Transducer:**
  - A PDA that outputs a string to a tape along with making transitions.
  - PDT = < Q, I, Γ, δ, q_0, Z_0, F, O, η>
  - PDT configuration c = <q, x, γ, z>, z is the string already written on the tape.

$$\forall x \in I^* \ (x \in L \wedge z = \tau(x) \Leftrightarrow c_0 = \langle q_0, x, Z_0, \varepsilon \rangle \ |\text{-}^*\text{-}\ c_F = \langle q, \varepsilon, \gamma, z \rangle \text{ and } q \in F)$$

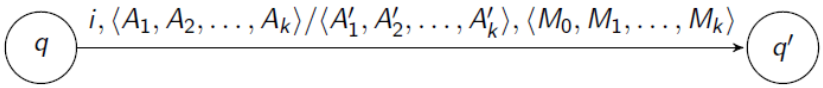
  - PDT acceptance condition: For all strings x that belongs the language constructed from the alphabet I, x belongs to some language L and z is the translation of x by the PDT if and only if there is a sequence of 0 or more moves that starts at the initial configurations c0 and ends at the final configuration cf.
  - c0 is when we are in the initial state q0, we still didn't read x, the stack is effectively empty and we haven't produced any output.
  - cf is when we are in some state q that is a final state, we read all x, stack currently has gamma in it, we have already written string z as the output.
- **Operations on PDA:**
  - Deterministic CFLs are closed under complement, but not under union, intersection, difference.
  - CFLs are closed under union, but not under complement, intersection, difference.
  - To complement a PDA:
    - Eliminate loops (make the PDA acyclic)
      - A PDA is acyclic if it doesn't have a spontaneous move.
      - You can convert any DPDA to an acyclic one.
    - Complete partial transition function.
    - Swap final and non-final states.
- **Automata Theory** is about the abstract mathematical machines and the computational problems that can be solved using them.
  - It studies different types of mathematical models of computation.
    - Sequential: FSA, PDA, TM.
    - Functional: Lambda calculus.
    - Concurrent: Petri nets, …
- **FSA Applications:**
  - Moore/Mealy Machines.
  - Lexical Analysis in compiler construction.
  - UML state machines.

Lecture 8:

- **Turing Machine:**
  - A simple -but conceptually important- mathematical model of a computer.
  - A theoretical machine not really meant for programming but for proofs and understanding.
  - TMs has the same expressive power as high-level programming language.
  - A TM can be seen as an abstract special purpose non programmable computer.
  - Uses tapes as memory, tapes are not destructive like stack.
    - They can be read multiple times.
  - Can simulate VNMs, but different in memory access.
    - TM can't have random access, only sequential.
    - This doesn't affect the computability, but the complexity.
  - It shows the results of computability theory.

- **TM - Informally:**
  - TMs has states and alphabet as other automaton.
    - Input/Output/Control device/Memory alphabet.
  - Tapes are infinite cell sequences, finite number of them is filled and the others contain a blank symbol '_'.
  - The machine does actions based on:
    - The current state of the machine.
    - One symbol read from the input tape.
    - K symbols, one read from each memory tape
  - Possible actions:
    - Change state
    - Write a symbol replacing the one written on (0 or more) memory tape(s).
    - Move (0 or more) of the k+1 heads (1 for input tape, k for memory tapes).
      - A head can move: One position Right or Left or Stand still.

- **TM - Graphically:**



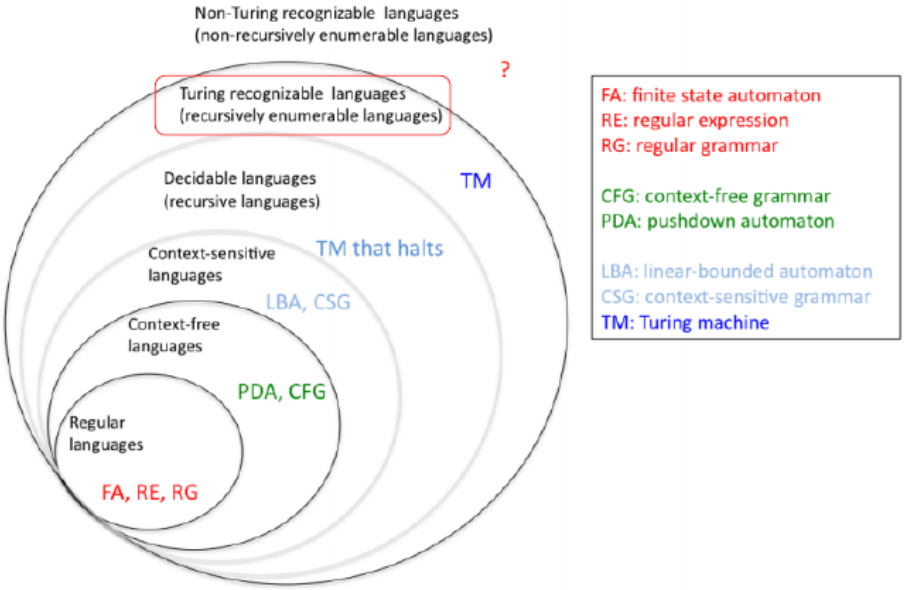
  - 'i' is the input symbol
  - A_j is the symbol read from the j-th memory tape
  - A_j' is the symbol replacing A_j
  - M_0 is the direction of the head of the input tape
  - M_j $(1 \leq j \leq k)$ is the direction of the head of the j-th memory tape

- **TM - Formally:**
  - TM = <Q, I, Γ, δ, q_0, Z_0, F>
  - Q: set of states I: input alphabet
  - Γ: memory alphabet, δ: transition function (Can't transit from final states).
  - $q_0 \in Q$: initial state, $Z_0 \in \Gamma$: initial memory symbol
  - $F \subseteq Q$: set of final states.
  - $\delta: (Q - F) \times (I \cup \{\_\}) \times (\Gamma \cup \{\_\})^k \rightarrow Q \times (\Gamma \cup \{\_\})^k \times \{R, L, S\}^{k+1}$

- **TM configuration (snapshot): is a k+2 tuple** $c = < q, x \uparrow y, \alpha_1 \uparrow \beta_1, \ldots, \alpha_k \uparrow \beta_k >$
  - q: current state.
  - x: current string written on the tape at the left of input reading head.
  - y: current string at the right of (including) head, concatenated with '_'.
  - α_j: current string at the left of head of memory tape j.
  - β_j: current string at the right of (including) head of memory tape j.
- **Acceptance in TM:**
  - A string $s \in I^*$ is accepted if $c_0 \vdash^* c_F$
  - $c_0 = < q_0, \varepsilon \uparrow s.\_, \varepsilon \uparrow Z_0, \ldots, \varepsilon \uparrow Z_0 >$
  - $c_F = < q, s' \uparrow y, \alpha_1 \uparrow \beta_1, \ldots, \alpha_k \uparrow \beta_k >, q \in F$
  - The language accepted by a Turing Machine 'T' can be defined as:
    - $L(T) = \{x | x \in I^* \wedge x \text{ is accepted by } T\}$
- **Von-Neumann Machine (VNM)** applies the stored program concept.
  - Instructions and data are stored in the same memory.
- **Harvard Architecture:** Instruction and data are stored in physically different memories.
- Regular languages (recognized by a FSA) < Context-Free Languages (recognized by a PDA) < Recursively Enumerable Languages (recognized by a TM).
- **Entscheidungsproblem (Decision problem by David Hilbert):**
  - Is there exists an algorithm that, given a statement in logic, tells whether it's universally valid or not?
    - Universally valid = can be deduced from logic axioms. Is it possible to find those axioms?
    - Is mathematics complete, consistent and decidable?
  - Turing and Church proved that there exists no algorithm.
- **Church-Turing Thesis:**
  - A function on the natural numbers can be effectively calculated iff it is computable by a Turing machine, or expressed through lambda calculus.
    - Any computation done by any physical process can be computed by the universal models described by Turing and Church.
- **Halting problem:**
  - Turing proved that it is not logically possible to write a computer program which can distinguish between programs that halt, and those that run forever.
- **Turing was the separator between idealism and materialism in mathematics**
  - **Idealism:** Human mind can access a higher domain of Platonic truths through mathematical reasoning.
  - **Materialism:** mathematical reasoning is bounded by the same constraints as TM which is a model of a human mathematician.

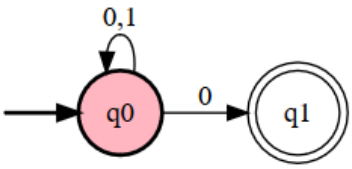
Lecture 9:

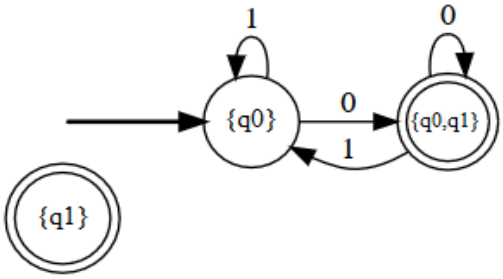
- **Operations on TM:**
  - TMs are closed under: intersection, union, concatenation, Kleene star.
    - A TM can simulate two other TMs in series or in parallel.
  - TMs are **not** closed under: complement, difference. $A - B = A \cap B^c$
    - A loop-free TM is closed under complement.
- **TM variants:**
  - A single unlimited tape that works as an input and memory.
    - Or it can have many tapes, to simulate parallel computations.
  - A tape can be 1D or multidimensional: a head for each dimension.
  - All these models are equivalent: they can recognize the same class of languages
    - They just provide different complexities.
- **Nondeterminism:**
  - NFSA and DFSA have **the same** expressiveness power.
  - NTM and DTM have **the same** expressiveness power.
  - NPDA **is more** expressive than DPDA.
    - NPDA can recognize all CFLs.
    - DPDA can only recognize deterministic CFLs.
- **Recursive languages:** languages that can be recognized by a TM that halts.
- **Recursively enumerable languages:** can be recognized by a TM that may halt or not.



- **Nondeterminism:**
  - A model of parallel computation where from one situation there are more than one possible action to be taken.
    - A switch case where there may be two cases that are true in the same time.
  - An abstraction to describe algorithm.
  - It is not a random/stochastic model.
- **Acceptance in nondeterministic machine:**
  - **Existential:** $x \in L \Leftrightarrow \delta^*(q_0, x) \cap F \neq \emptyset$: at least one of the resulting states applying the input is a final state.
  - **Universal:** $x \in L \Leftrightarrow \delta^*(q_0, x) \subseteq F$: all resulting states are final states.

- **Nondeterministic FSA**
  - A tuple: <Q, I, δ, q_0, F> where δ: Q × I → P(Q): Power set on Q.
  - δ^* produces a set of states instead of one state.
  - Has the same expressiveness power as DFSA
    - Given a NDFA, one can construct an equivalent DFSA.
      - If A_{ND} = <Q, I, δ, q_0, F> then A_D = <Q_D, I, δ_D, q_{0D}, F_D>
        - $Q_D = P(Q)$
        - $\delta_D(q_D, i) = \bigcup_{q \in q_D} \delta(q, i)$
        - $q_{0D} = \{q_0\}$
        - $F_D = q_D \mid q_D \in Q_D \wedge q_D \cap F \neq \emptyset$ }
    - Example: NFSA that accepts all binary strings that ends with 0 and its equivalent DFSA using this strategy.

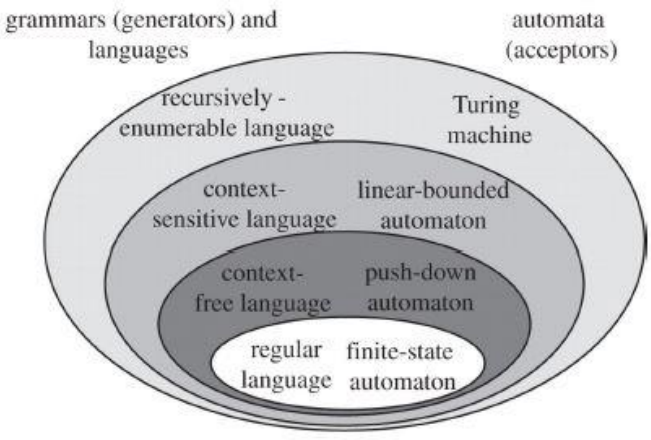




    - NFSA is useful because it is easier to design
      - Number of states is can be exponentially less.
      - A NFSA with n states can have its equivalent DFSA with 2^n states in the worst case.

Lecture 10:

- **Nondeterministic TM**
  - Has the same properties as TM but the transition function is modified.
    - $\delta: (Q\text{-}F) \times I \times \Gamma^k \rightarrow P(Q \times \Gamma^k \times \{R,L,S\}^{k+1})$
  - Acceptance in NDTM is existential (see lec. 9)
  - Computations can be depicted as a tree with 3 possibilities
    - Tree has a leaf node with accepting state (TM halts and accepts)
    - Tree has a leaf node with no accepting state (TM halts and rejects)
    - Tree has no leaf node (TM doesn't halt)
      - In that case, to know whether a string is accepted or not we need to perform a Breadth-First Visit, because a Depth-First Visit may not terminate.
  - We can build a DTM that simulates a NDTM, therefore, both of them have the same expressiveness power.
- **Nondeterministic PDA**
  - Has the same properties as DPDA, but it doesn't have the restriction on ε move.
    - The restriction was that, from any state, either the epsilon move applies, or the regular move applies, but not both.
  - We can more than one transition from the same state, with the same input, having the same symbol on top of the stack.
  - **Formally:** $\delta: Q \times (I \cup \{\varepsilon\}) \times \Gamma \rightarrow P_F (Q \times \Gamma^*)$
    - P_F indicates the finite subset of $Q \times \Gamma^*$
  - NPDA is more expressive than DPDA, for example it can recognize the language L.
    - $L = \{ a^n b^n \mid n \geq 1\} \cup \{ a^n b^{2n} \mid n \geq 1 \}$
  - NPDAs are closed under union,
  - NPDAs are not closed under intersection and complement.
    - Nondeterministic machines cannot be complemented in general.
- **Generative grammars.**
  - Automaton are **operational** models of languages; they receive an input and **process** it.
    - "Process" here means recognize or translate.
  - Grammars are **generative** models of languages; they describe how to generate a language by providing a set of rules for building the languages, they produce strings.
- **Chomsky hierarchy:**
  - describes a hierarchy of languages and the corresponding acceptors.
- **Formal grammar:**
  - Generates strings of a language through a **rewriting** process.
    - Rewriting is the process of replacing sub-terms of a formula, with other terms.



the traditional Chomsky hierarchy

- **Grammar/syntax:**
  - A set of rules to build the phrases of a language.
    - It applies to any notion of language (natural, artificial…)
  - It consists of:
    - **A main object**: initial symbol
    - **Composing objects:** nonterminal symbols
    - **Base elements:** terminal symbols
    - **Refinement rules:** productions
  - A linguistic rule describes a **main object** as a sequence of **composing objects**, each composing object is **refined** by replacing/rewriting it with more detailed objects until a sequence of **base elements** (that cannot be further refined) is obtained
  - **Examples:**
    - natural languages and programming languages are both explained through a specific grammar.
  - **Formally:**
    - **A grammar is a tuple < V_N, V_T, P, S > where**
      - V_N is the **nonterminal alphabet** (or vocabulary)
      - V_T is the **terminal alphabet** (or vocabulary)
      - $V = V_N \cup V_T$
      - $S \in V_N$ is a particular element of V_N called **axiom** / **initial symbol.**
      - $P \subseteq <\alpha, \beta>$ is the (finite) set of rewriting rules or **productions.**
        - $\alpha \in V^*. V_N . V^*$, $\beta \in V^*$ (dot is the string concatenation).
        - Production is indicated as $\alpha \rightarrow \beta$
        - α is a sequence of symbols including at least one nonterminal symbol.
        - β is a (potentially empty) sequence of terminal and/or non-terminal symbols.
        - Immediate derivation relation is the relation from α to β
          - It means that the string β is derived from α
    - **A grammar G = < V_N, V_T, P, S > generates a language on the alphabet V_T.**
  - **Example of a grammar with derivation:**
    - $V_N = \{S, A, B, C, D\}$
    - $V_T = \{a, b, c\}$
    - S is the initial symbol.
    - $P = \{S \rightarrow AB, BA \rightarrow cCD, CBS \rightarrow ab, A \rightarrow e\}$
    - aa**BA**S → aa**cCD**S
    - bc**CBS**Add → bc**ab**Add
  - L(G) denotes **the language generated by the grammar G**, it's the set of all string consisting of only terminal symbols (V_T) that can be derived from the initial symbol (S) in any number of steps.
    - Usually we use capital letter for elements in V_N, and small letter for V_T
    - **Example:** The grammar G
      - G = < {S, A, B}, {a, b, 0}, P, S>
      - P = { S → aA, A → aS, S → bB, B → bS, S → 0 }
    - Generates the language L(G) = {aa, bb}*.0

Lecture 11:

- **Alternative definitions for a grammar:**
  - A grammar can be seen as a device that enumerates the sentences of a language.
  - A grammar of L can be seen as a function whose range is exactly L
- Grammars are classified into 4 types; each one has a minimal automaton that can recognize it.

Chomsky hierarchy	Grammars (equivalent names)	Languages	Minimal automaton
Type-0	Unrestricted (general)	Recursively enumerable	Turing machine
Type-1	Context-sensitive	Context-sensitive	**(Linear bounded automaton)**
Type-2	Context-free	Context-free	NDPDA
Type-3	Regular	Regular	FSA

- **Type-x can also be considered type-y if y < x**
- **A, B = nonterminal characters, α, β, γ = strings of terminals and/or non-terminals.**
- **Type-3**: Rewrite a nonterminal as a terminal followed/preceded by at most one nonterminal, but not both. (left-linear grammar XOR right-linear grammar). ($A \rightarrow \alpha B$ XOR $A \rightarrow B\alpha$)
  - The rule $S \rightarrow \varepsilon$ is allowed if S does not appear on the right side of any production rule.
- **Type-2**: Rewriting of A doesn't depend on its surroundings/context ($A \rightarrow \gamma$)
  - $|A| = 1$, $\gamma \in V^*$ (Can also be written in Backus-Naur Form (BNF), **check lab 11**)
- **Type-1**: Context sensitive ($\alpha A \beta \rightarrow \alpha \gamma \beta$).
  - LHS and RHS may be surrounded by any string of terminals / non-terminals, $\gamma \neq \varepsilon$
  - The rule $S \rightarrow \varepsilon$ is allowed if S does not appear on the right side of any production rule.
- **Type-0**: No restrictions, includes all other types ($\alpha \rightarrow \beta$)
  - **α:** any string of terminals and non-terminals (except ε)
  - **β:** any string of terminals and non-terminals.
- **Examples:**

Type-0

$VN = \{S, T, C, P\}$
$VT = \{a, b\}$
$P = \{S \rightarrow T\ E$
$T \rightarrow aTa \mid bTb \mid C$
$C \rightarrow CP$
$Paa \rightarrow aPa$
$Pab \rightarrow bPa$
$Pba \rightarrow aPb$
$Pbb \rightarrow bPb$
$PaE \rightarrow Ea$
$PbE \rightarrow Eb$
$CE \rightarrow \varepsilon$
}

Type-1

$V_N = \{S, A, B\}$
$V_T = \{a,b,c\}$
$P = \{S \rightarrow abc \mid aAbc,$
$Ab \rightarrow bA$
$Ac \rightarrow Bbcc$
$bB \rightarrow Bb$
$aB \rightarrow aa$
$aB \rightarrow aaA$
}

$L=\{a^n b^n c^n \mid n \geq 1\}$

Type-2

$V_N = \{S\}$
$V_T = \{a,b\}$
$P = \{S \rightarrow aSb \mid \varepsilon\}$

$L=\{a^n b^n \mid n \geq 0\}$

Type-3

$V_N = \{S\}$
$V_T = \{a\}$
$P = \{S \rightarrow aS \mid \varepsilon\}$ (right linear)

$L=\{a^n \mid n \geq 0\}$

- **Linear-Bounded Automaton (LBA):**
  - The minimum automaton that recognizes context-sensitive languages.
  - A Turing machine with only one tape for reading input and processing string.
  - The tape is finite and has length = |x| + 2, where |x| is the length of input string.
    - The two extra places contain a special left and right end symbols that the head can't go beyond.
- **Building a Regular Grammar from a Finite State Automaton.**
  - If A = < Q, I, δ, q_0, F > then G = <V_N, V_T, P, S> such that:
    - V_N = Q, V_T = I, S = <q_0>
    - ∀δ(q, i) = q'
      - <q> → i<q'> ∈ P
      - q' ∈ F ⇒ <q'> → ε ∈ P
    - δ*(q, x) = q' ⇔ <q> ⇒* x<q'>
      - ⇒* means that LHS will eventually lead to RHS, not necessarily by direct derivation.
- **Building a Finite State Automaton from a Regular Grammar.**
  - If G = <V_N, V_T, P, S> then A = < Q, I, δ, q_0, F > such that:
    - Q = V_N ∪ {q_F}, I = V_T, <q_0> = S, F = {q_F}
    - ∀A → bC ∈ P ⇒ δ(A, b) = C
    - ∀A → b ∈ P ⇒ δ(A, b) = q_F
- **Post Correspondence Problem (by Emil Post)**
  - Given two lists A, B: A = w_1, w_2, …, w_k, B = x_1, x_2, …, x_k, **determine whether there is a sequence of m ≥ 1 integers i_1, i_2, …, i_m such that:**
    - $w_1, w_{i_1}, w_{i_2}, \dots, w_{i_m} = x_1, x_{i_1}, x_{i_2}, \dots, x_{i_m}$.
    - (w_i, x_i) us called a correspondence pair.
  - **Example (On the right).**
  - This problem is **undecidable** 😊
    - Like the halting/decision problem.
    - But it's simpler to express.

	A	B
i	w_i	x_i
1	11	1
2	1	111
3	0111	10
4	10	0

This PCP instance has a solution: 3, 2, 2, 4:
$w_1w_3w_2w_2w_4 = x_1x_3x_2x_2x_4$ = 1101111110

- **Two questions in computer science:**
  - **Mathematical** question: What can be computed? What cannot be computed?
  - **Engineering** question: How can we build computers?
- **Computability Theory (See first line of lec1)**
  - Is about the mathematical question of computer science.
  - Applied in software verification.
  - Limits of computation are
    - In common with the human computation.
    - Bounded by laws of physics and mechanics.
  - Computability theory is about two aspects:
    - Do there exist computing formalisms more powerful than TMs? (Church-Turing thesis)
    - Can we always solve problems by means of some mechanical device? (Halting problem and undecidability).

Lecture 12:

- **Church-Turing Thesis:**
  - There is no formalism to model any mechanical calculus that is more powerful than the TM or equivalent formalisms.
    - Quantum computing doesn't break this.
  - Any algorithm can be coded in terms of a TM (or an equivalent);
- **Algorithmic enumeration:**
  - Given a set, we can enumerate its elements using a function E: S $\rightarrow \mathbb{N}$
    - Constructing E require having a bijection (one-to-one correspondence) between S and $\mathbb{N}$.
    - E(M) where M is a TM is called the Gödel number of M
    - E is called a Gödelization.
  - Turing machines can be enumerated in the same way (effective enumeration).
    - They can also be enumerated lexicographically.
    - Number of Turing machine with |Q| states and cardinality of alphabet |A| is equal to $(|Q||A|^2 + 1)^{|Q||A|}$
    - We can use this fact to enumerate all TM's staring from the ones with one state then two states, and so on…
- **Two important questions:**
  - **Can TMs model programmable computers?** YES
    - A Universal Turing Machine (UTM) computes the function $g(y, x) = f_y(x)$ which is the function computed by the y-th TM on input x.
    - UTM takes two inputs, the program and the data (like a VNM)
    - UTM can be considered a programmable computer with a program stored in memory (y) and works on an input (x), while TM can be considered a non-programmable computer with a single built-in program.
  - **Can TMs compute all functions from $\mathbb{N}$ to $\mathbb{N}$?** NO
  - There are functions that cannot be computed algorithmically. They are a lot more than the functions that can be computed ($\aleph_0 < 2^{\aleph_0}$)
    - Cardinality of the set of all functions from $\mathbb{N}$ to $\mathbb{N} \geq$ Cardinality of the set of all functions from $\mathbb{N}$ to {0, 1} = Cardinality of all real numbers = $2^{\aleph_0}$
- **The Halting Problem:**
  - Given a program and an input to the program determine if the given program will eventually stop with this particular input.
  - Formally, is there a TM that computes g:
    - **$g(y,x) = 1$ if $f_y(x) \neq \perp$, $g(y,x) = 0$ if $f_y(x) = \perp$** (defined/ the TM halts)
  - **Undecidable** again, because programs that analyze programs can be made to analyze themselves, leading to the impossibility (contradiction).
  - As a result, a TM that decides whether another TM will halt or not on a given input does not exist.
  - *** check HP undecidability proof in lecture slides*.**

- **Some proof techniques:**
  - **Direct proof**
    - by axioms, theorems…
  - **Proof by induction**
    - base case, inductive case…
  - **Constructive proof**
    - Provide an example (or counterexample)
  - **Proof by contradiction**
    - Reductio ad absurdum.
  - **Proof by diagonalization**
    - Often used in computability proofs.
    - First used by Cantor to prove that $|\mathbb{R}| \geq |\mathbb{N}|$
    - Also used to prove the undecidability of the Halting Problem.
- **Russel's paradox**:
  - Contradicts what Cantor said that "any definable collection is a set".
  - Let R be the set of all sets that are not a member of themselves, if R is in R, then it shouldn't be in R by definition, but it is😊 → contradiction.
  - Zermelo-Fraenkel set theory deals with Russel's paradox.
- **Gödel's First Incompleteness Theorem**
  - An **Effective**, **Consistent** and **Rich** logical system is not **Complete**.
    - **Effective**: axioms and deduction rules can be followed by a computer.
    - **Consistent**: A and not A, cannot be both proven.
    - **Rich**: containing at least arithmetic
    - **Complete**: you can either prove A or not A.

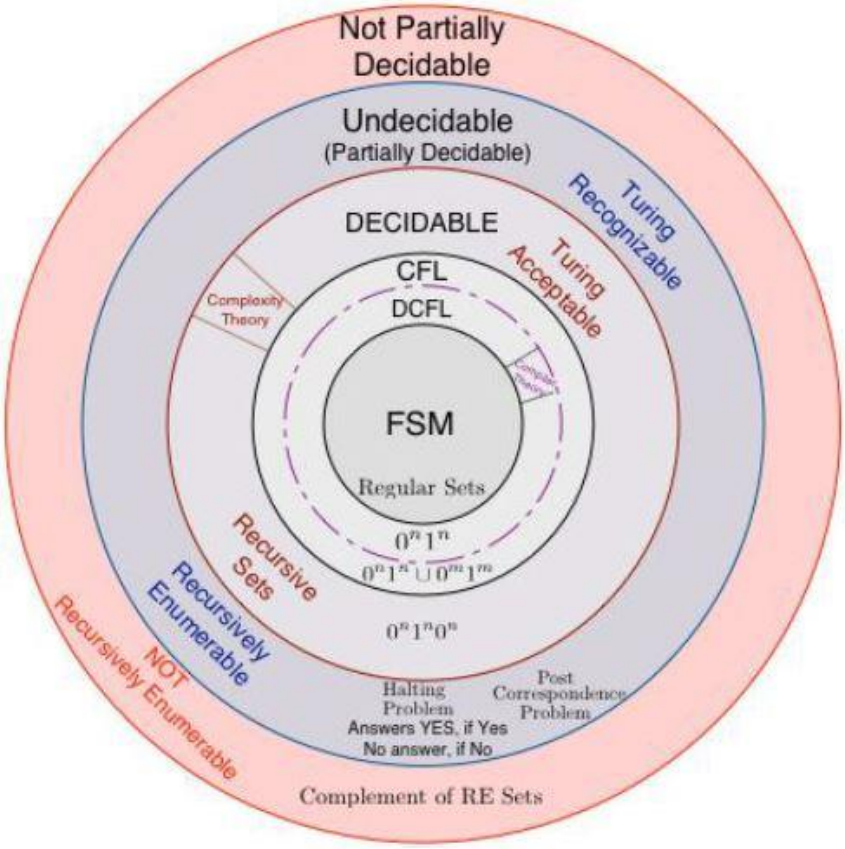
Lecture 13:

- **A decision problem is the problem that has one answer "yes" or "no"**
- **Decidability**
  - Decidable problem means there is an algorithm that can provide the correct answer.
  - Semi-decidable problem means there exists an algorithm that says "yes" when the answer is "yes". It may loop infinitely if the answer is "no".
  - **Examples:** Halting problem, checking runtime errors in programs.
    - "Testing can prove the presence of errors, not their absence".
- **Recursively enumerable (RE) languages:**
  - A set S is recursive (or decidable) if and only if its characteristic function is computable.
  - Characteristic function of a set S determines whether an element x belong to S.
    - c_S (x) = if x ∈ S then 1 else 0
  - S is recursively enumerable (or semi-decidable) if
    - S = ∅.
    - Or S is the image of a total, computable function g.
      - $S = I_{gS} = \{ x \mid x\ g_S(y), y \in \mathbb{N} \} \Rightarrow S = \{ g_S(0), g_S(1), g_S(2), \dots \}$.
    - In that case we can know if x ∈ S, even if S is infinite.
    - but we can't know if x ∉ S and S is infinite.
  - **Two results:**
    - If S is recursive, then it's also recursively enumerable.
    - S is recursive ⇔ S and $\mathbb{N}$ - S are both recursively enumerable
      - The class of decidable sets is closed under complement.
- **Non-recursively enumerable languages:**
  - The set of all total computable functions is non-recursively enumerable.
    - Can be proved by diagonalization.
    - There is no automata that can recognize this languages.
- **Kleene's fixed-point theorem:**
  - Let t be a total and computable function Then it is always possible to find an integer p such that $f_p = f_{t(p)}$ (Function f p is called a fixed point of t).
    - For any total computable function f, there is a number p such that both p and t(p) indicate the same computable function.
  - In other words, a function F will have at least one fixed point (a point x for which F(x) = x), under some conditions on F that can be stated in general terms.
- **Rice's theorem: *check the proof in lecture slides***
  - Every non-trivial property of the recursively enumerable languages is undecidable.
    - Trivial = applies on all TM's or none of them.
- **Formally:**
  - Let F be a set of all computable functions.
  - Let S be the set of all TMs (indices) that compute the functions of F.
    - $S = \{ x \mid f_x \in F \}$
  - Then, S is decidable ⇔ if F = ∅ or F is the set of all computable function.
- **Applications:**
  - Program correctness: does TM_x computes a specific function
  - Program equivalence: does the machines TM_x and TM_y compute the same functions.
  - Does the program compute a specific kind of function only?

- **Syntactic vs semantic properties of programs:**
  - A **syntactic property** is purely about program structure
    - Example: Does the program contain an if-then-else statement?
  - A **semantic property** is about program's behavior
    - Example: Does the program terminate for all inputs?
- **Rice's theorem, in other words:**
  - States that all non-trivial semantic properties of programs are undecidable.
  - There is an endless list of interesting problems whose undecidability follows trivially from Rice s theorem.
  - For any chosen set F = {g} by Rice's theorem it is not decidable whether a generic given TM computes g or not.
  - For every non trivial property of partial functions no general and effective method can decide whether an algorithm computes a partial function with that property.
  - Any interesting property of program behavior is undecidable (program analysis).
  - It is impossible to fully automatize software verification.

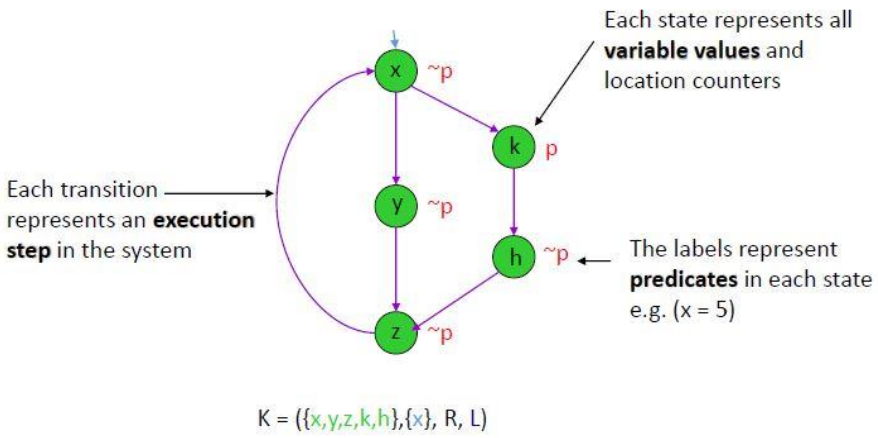
Lecture 14:

- **The BIG picture:**

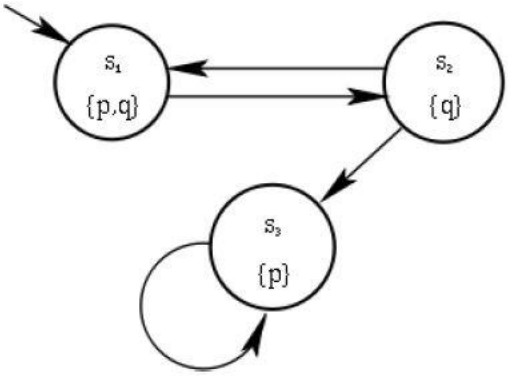


- **Formal verification:**
  - Using mathematical arguments to test software and find bugs during design phase.
- **Program verification problem:**
  - Given a program P and a specification S, determine whether every execution of P, for any value of input arguments satisfies S.
  - In general, the problem is again **undecidable**.
  - It may become decidable, if we add some restriction to P and S.
    - For example, if P is a finite state, but real programs are not usually finite states.
- **Model checkers:**
  - Formal verification tools, that are used for doing massive (but often simple) case analysis that are hard for the human to do.
- **Software Model Checking:**
  - Techniques used to automatically verify real programs based on finite state models of them.
  - Protocols (digital circuits, more recently software) are modeled as state-transition systems.
  - Specifications are a formula f in propositional temporal logic
  - Verification procedure is an exhaustive (but efficient) search of the state space of the design to see if the model satisfies f.

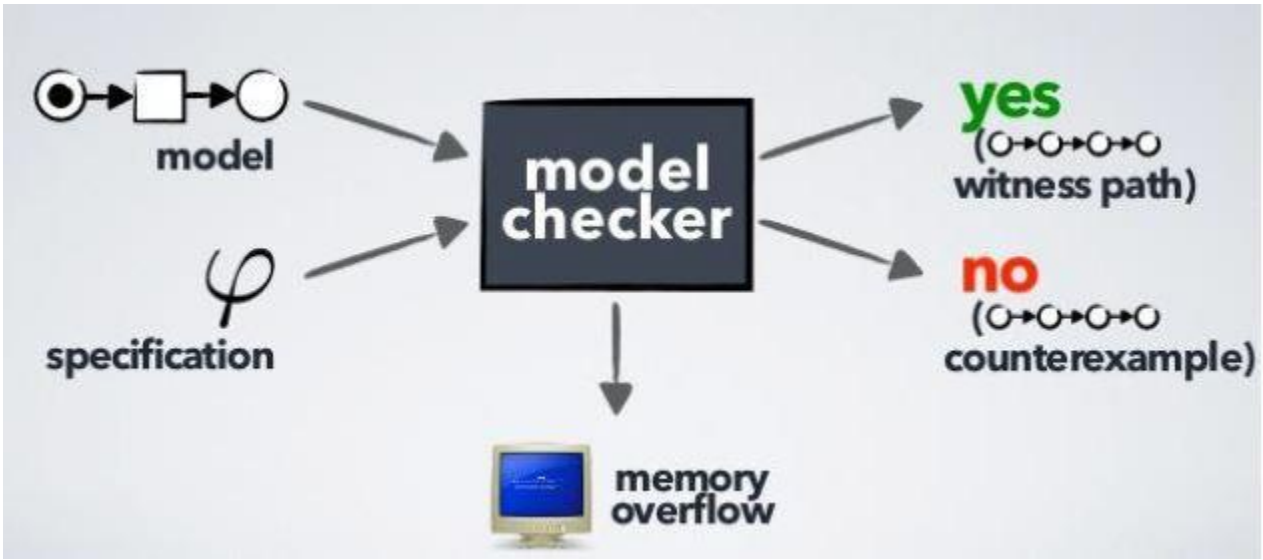
- **Labeled state graph (Kripke structure).**



- **Formal example:** M = (S, I, R, L)**:**
  - S = {s1, s2, s3} – set of states
  - I = {s1} – Initial state
  - R = {(s1, s2), (s2, s1), (s2, s3), (s3, s3)} – Relation on S that defines all possible transitions.
  - L = {(s1, {p, q}), (s2, {q}), (s3, {p})} – Labels for states (Holds the predicates attached to states)



- **Model checking process:**



Tutorials:

Useless definitions from Philosophy:

- Complete Boolean algebra is a **Boolean algebra** with every subset has a LUB
- Boolean algebra is a **distributive lattice** in which every element has a **complement**.
- A **complement** of an element p of a lattice is an element p0 such that p ^ p0 is the least element (0) and p v p0 is the greatest element (1).
- **Distributive lattice** is a **lattice** that has: A v (B ^ C) = (A v B) ^ (A v C)
- A **lattice** is a **poset** that has a least and a greatest element and such that any two elements have a least upper bound and a greatest lower bound.
- A **p**artially **o**rdered **set** is a set with a relation (≤) on it.

Regular expressions:

- **Inductive definition of the set RegExp of regular expressions over an alphabet Σ:**
  - **Basis:**
    - The symbol ∅ is a regular expression;
    - The symbol ε is a regular expression;
    - Each symbol σ of Σ is a regular expression.
  - **Induction:**
    - Let r and s be two regular expressions, then
      - (r . s) is a regular expression;
      - (r | s) is a regular expression;
      - r* is a regular expression.
- **The language denoted by a regular expression**
  - For any regular expression E over Σ, we define, by induction on E, the language L(E) ⊆ Σ* denoted by E:
  - **Basis:**
    - L(∅) = ∅
    - L(ε) = {ε}
    - L(σ) = {σ} for each σ ∈ Σ.
  - **Induction:**
    - Let r and s be two regular expressions, then
      - L((r . s)) = {ww' | (w ∈ L(r) ^ w' ∈ L(s))}
      - L((r | s)) = L(r) ∪ L(s)
      - L(r*) = L(r)*.
  - **Example:**
    - (0*1*)*000(0 | 1)* describes the set of strings containing 3 consecutive 0's
    - (1 | ε)(00*1)*0* describes the set of strings that don't have consecutive 1's
- **Lookup (lab10):**
  - **Kleene's Algorithm:** Transforms FSA to RegExp
  - **Thompson construction:** Transforms RegExp to FSA

Lambda calculus:

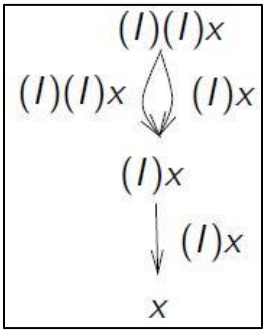
- **Before the introduction of TM's, two other formalisms were introduced:**
  - Partial recursive functions and Lambda Calculus
- **Church introduced the λ-calculus in 1936 to solve the decision problem (check lec.8).**
  - He proved that the set of universally valid formulas is non-recursive.
- **Informally:** The λ-calculus is a functional programming language. It can be seen as:
  - A formalization of effectively computable functions.
  - A paradigm for programming languages (ex. Scheme programming language)
  - A formalization of mathematics on the notion of mapping.
  - An internal language of Cartesian closed categories.
  - A way to translate mathematics into programs (ex. Coq)
- **The objects of the λ-calculus are functions. We deal only with:**
  - Variables, Applications, Abstractions
- **Lambda expression is valid only when it consists of lambda terms**
  - The set of all lambda terms Λ can be defined as
    - $\Lambda = V \mid \lambda V.\Lambda \mid (\Lambda)\Lambda$.
    - 'V' is set of all variable that we can use
    - '|' mean OR "obviously".
    - λV.Λ means: define a function "with no name" that takes V as an argument and returns the lambda term Λ.
    - (Λ)Λ means: apply the lambda term Λ on the lambda term Λ.
- **Carvalho temporary grammar to definitive grammar:**
  - @(f, x) = f x = (f)x "the application of function f on parameter x".
  - x → exp = λx.exp "the function, takes an argument x and returns the expression exp".
- **Natural numbers using lambda calculus.**
  - Church encodes natural numbers as functions that take as an argument a function and returns a function.

Number	Function definition	Lambda expression
0	$0\ f\ x = x$	$0 = \lambda f.\ \lambda x.\ x$
1	$1\ f\ x = f\ x$	$1 = \lambda f.\ \lambda x.\ f\ x$
2	$2\ f\ x = f\ (f\ x)$	$2 = \lambda f.\ \lambda x.\ f\ (f\ x)$
3	$3\ f\ x = f\ (f\ (f\ x))$	$3 = \lambda f.\ \lambda x.\ f\ (f\ (f\ x))$
⋮	⋮	⋮
n	$n\ f\ x = f^n\ x$	$n = \lambda f.\ \lambda x.\ f^{\circ n}\ x$

  - We can also define some basic functions on natural numbers as λ-terms
    - Pred(n) = n – 1
    - Succ(n) = n + 1
    - Plus(m, n) = m + n
    - Minus(m, n) = m – n
    - Mult(m , n) = m * n
    - Exp(m, n) = m^n

$$\text{pred} \equiv \lambda n.\ \lambda f.\ \lambda x.\ n\ (\lambda g.\ \lambda h.\ h\ (g\ f))\ (\lambda u.\ x)\ (\lambda u.\ u)$$
$$\text{succ} \equiv \lambda n.\ \lambda f.\ \lambda x.\ f\ (n\ f\ x)$$
$$\text{plus} \equiv \lambda m.\ \lambda n.\ \lambda f.\ \lambda x.\ m\ f\ (n\ f\ x)$$
$$\text{minus} \equiv \lambda m.\ \lambda n.\ (n\ \text{pred})\ m$$
$$\text{mult} \equiv \lambda m.\ \lambda n.\ \lambda f.\ \lambda x.\ m\ (n\ f)\ x$$
$$\text{exp} \equiv \lambda m.\ \lambda n.\ n\ m$$

- **Free and bound variables:**
  - λx.xy has x as a bound variable and y as a free variable because the expression xy depends on x and y where x is the parameter, but y is not.
    - (λx.x)**x** "here x is bound, but **x** is free".
  - If there are several λ's in the term, then a variable can be bound to a specific lambda but not the others.
    - (λx.x)(λy.xy) "x is bound to the first lambda, but not the second".
  - A λ-term is closed if it has no free variables.
- **Reduction operations on λ-terms:**
  - **α-equivalence**
    - Two lambda terms are **α-equivalent** if they represent the same term with different variable names.
      - Free variables shouldn't be renamed, it will change the expression.
  - **β-reduction**
    - The operation which reduces the lambda terms by actually applying the functions on their arguments, replacing every occurrence of the parameter with the argument.
      - That lambda term is called a RedEx = Reducible Expression.
    - The reduction operation is denoted as u[v/x].
      - It means: the expression u in which every occurrence of x is replaced with v.
    - A lambda term 't' is β-normalizable ⇔ ∃ a β-normal term t' such that t β* t'.
      - The λ-term that cannot be reduced is called β-normal.
      - β* is the reflexive transitive closure on β.
      - t β* t' if we can obtain t' from t by 0 or more β-reduction.
    - **Example:**
      - (λx.x)y reduces by β-reduction to y ⇔ (λx.x)y β y
- **Reduction graph**
  - A directed weighted graph in which:
    - The vertices are λ-terms.
    - The edges connect t → t' iff t β* t'
    - The weight is the redex used to reduce t to t'
  - **Example:**
    - the reduction graph of (λz.z)(λy.y)x, set I = λs.s



- **Consider this λ-term: (λz.(z)z)λz.(z)z**
  - **Meaning:**
    - Define a function that takes a function z as a parameter and returns z applied on itself. See that function you just created? apply it on itself 😊
  - This term has a cycle in its reduction graph and thus, it is not β-normalizable,
    - However, the graph is finite.
- **Consider this λ-term: (λx.(x)x)λy.(I)(y)y, I = λs.s**
  - This term produces an infinite graph. However, this doesn't mean that it cannot be normalized. These terms can be normalized sometimes.
- **Church-Rosser Theorem:** The relation β* has the **diamond property**, but the relation β doesn't, therefore, a λ-term can have many possible β-reductions, but at most 1 β-normal form.
  - A binary relation R on some set S is said to have the diamond property if
    - For any t, t_1, $t_2 \in S \mid t\ R\ t_1 \wedge t\ R\ t_2 \Rightarrow \exists\ t_3 \in S \mid t_1\ R\ t_3 \wedge t_2\ R\ t_3$