MODULE 5

1. **Turing Machines**
2. **The Halting Problem**
3. **The Universal language**
4. **A Church- Turing thesis**
5. **Linear Bounded Automata.**

1. Turing Machines (TM)

- **Generalize the class of CFLs:**

- Recursively enumerable languages are also known as *type 0* languages.
- Context-sensitive languages are also known as *type 1* languages.
- Context-free languages are also known as *type 2* languages.
- Regular languages are also known as *type 3* languages.

- TMs model the computing capability of a general purpose computer, which informally can be described as:
  - Effective procedure
    - Finitely describable
    - Well defined, discrete, "mechanical" steps
    - Always terminates
  - Computable function
    - A function computable by an effective procedure

- TMs formalize the above notion.

1.1 Deterministic Turing Machine (DTM)

- Two-way, infinite tape, broken into cells, each containing one symbol.
- Two-way, read/write tape head.
- Finite control, i.e., a program, containing the position of the read head, current symbol being scanned, and the current state.
- An input string is placed on the tape, padded to the left and right infinitely with blanks, read/write head is positioned at the left end of input string.
- In one move, depending on the current state and the current symbol being scanned, the TM 1) changes state, 2) prints a symbol over the cell being scanned, and 3) moves its' tape head one

cell left or right.

- Many modifications possible.

1.2 Formal Definition of a DTM

– A DTM is a seven-tuple:

$M = (Q, \Sigma, \Gamma, \delta, q_0, B, F)$

Q	A <u>finite</u> set of states
Γ	A <u>finite</u> tape alphabet
B	A distinguished blank symbol, which is in Γ
Σ	A <u>finite</u> input alphabet, which is a subset of Γ– {B}
q_0	The initial/starting state, q_0 is in Q
F	A set of final/accepting states, which is a subset of Q
δ	A next-move function, which is a *mapping* from Q x Γ –> Q x Γ x {L,R}

Intuitively, δ(q,s) specifies the next state, symbol to be written and the direction of tape head movement by M after reading symbol s while in state q.

- **Example #1:** $\{0^n1^n \mid n >= 1\}$

	0	1	X	Y	B
q_0	(q_1, X, R)	-	-	(q_3, Y, R)	-
q_1	$(q_1, 0, R)$	(q_2, Y, L)	-	(q_1, Y, R)	-
q_2	$(q_2, 0, L)$	-	(q_0, X, R)	(q_2, Y, L)	-
q_3	-	-	-	(q_3, Y, R)	(q_4, B, R)
q_4	-	-	-	-	-

– **Example #1:** $\{0^n1^n \mid n >= 1\}$

	0	1	X	Y	B
q_0	(q_1, X, R)	-	-	(q_3, Y, R)	-
q_1	$(q_1, 0, R)$	(q_2, Y, L)	-	(q_1, Y, R)	-
q_2	$(q_2, 0, L)$	-	(q_0, X, R)	(q_2, Y, L)	-
q_3	-	-	-	(q_3, Y, R)	(q_4, B, R)
q_4	-	-	-	-	-

- The TM basically matches up 0's and 1's
- q_1 is the "scan right" state
- q_2 is the "scan left" state
- q_4 is the final state

– **Example #2:** {w | w is in {0,1}* and w ends with a 0}

0
00
10
10110
Not ε

$Q = \{q_0, q_1, q_2\}$
$\Gamma = \{0, 1, B\}$
$\Sigma = \{0, 1\}$

$F = \{q_2\}$

	0	1	B
q_0	$(q_0, 0, R)$	$(q_0, 1, R)$	(q_1, B, L)
q_1	$(q_2, 0, R)$	-	-
q_2	-	-	-

- q_0 is the "scan right" state
- q_1 is the verify 0 state

– **Definition:** Let M = (Q, Σ, Γ, δ, q_0, B, F) be a TM, and let w be a string in Σ*. Then w is *accepted* by M iff

$$q_0w \vdash\!\!-^* \alpha_1 p \alpha_2$$

Where p is in F and α_1 and α_2 are in Γ*

– **Definition:** Let M = (Q, Σ, Γ, δ, q_0, B, F) be a TM. The *language accepted by M*, denoted L(M), is the set

L={w | w is in Σ* and w is accepted by M}

In contrast to FA and PDAs, if a TM simply *passes through* a final state then the string is accepted.

- Given the above definition, no final state of an TM need have any exiting transitions. *Henceforth, this is our assumption*.
- **If x is not in L(M) then M may enter an infinite loop, or halt in a non-final state**.
- Some TMs halt on all inputs, while others may not. In either case the language defined by TM is still well defined.

– **Definition:** Let L be a language. Then L is *recursively enumerable* if there exists a TM M such that L = L(M).

  – If L is r.e. then L = L(M) for some TM M, and
    •If x is in L then M halts in a final (accepting) state.
    •If x is not in L then M may halt in a non-final (non-accepting) state, or loop forever.

– **Definition:** Let L be a language. Then L is *recursive* if there exists a TM M such that L = L(M) and M halts on all inputs.

  – If L is recursive then L = L(M) for some TM M, and
    •If x is in L then M halts in a final (accepting) state.
    •If x is not in L then M halts a non-final (non-accepting) state.

  – The set of all recursive languages is a subset of the set of all recursively enumerable languages

  – Terminology is easy to confuse: A *TM* is not recursive or recursively enumerable, rather a *language* is recursive or recursively enumerable.

– **Observation:** Let L be an r.e. language. Then there is an infinite list M_0, M_1, … of TMs such that L = L(M_i).

– **Question:** Let L be a recursive language, and M_0, M_1, … a list of all TMs such that L = L(M_i), and choose any i>=0. Does M_i always halt?

Answer: Maybe, maybe not, but *at least one in the list does*.

– **Question:** Let L be a recursive enumerable language, and M_0, M_1, … a list of all TMs such that L = L(M_i), and choose any i>=0. Does M_i always halt?

  Answer: Maybe, maybe not. Depending on L, none might halt or some may halt.

– If L is also recursive then L is recursively enumerable.

Question: Let L be a recursive enumerable language that is not recursive (L is in r.e. – r), and M_0, M_1, … a list of all TMs such that L = L(M_i), and choose any i>=0. Does M_i always halt?
Answer: No! If it did, then L would not be in r.e. – r, it would be recursive.

- **Let M be a TM.**
  - Question: Is L(M) r.e.?
    Answer: Yes! By definition it is!
  - Question: Is L(M) recursive?
    Answer: Don't know, we don't have enough information.
  - Question: Is L(M) in r.e – r?
    Answer: Don't know, we don't have enough information.
- **Let M be a TM that halts on all inputs:**
  - Question: Is L(M) recursively enumerable?
    Answer: Yes! By definition it is!
  - Question: Is L(M) recursive?
    Answer: Yes! By definition it is!
  - Question: Is L(M) in r.e – r?
    Answer: No! It can't be. Since M always halts, L(M) is recursive.
- **Let M be a TM.**
  - As noted previously, L(M) is recursively enumerable, but may or may not be recursive.
  - Question: Suppose that L(M) is recursive. Does that mean that M always halts?
    Answer: Not necessarily. However, some TM M' must exist such that L(M') = L(M) and M' always halts.
  - Question: Suppose that L(M) is in r.e. – r. Does M always halt?
    Answer: No! If it did then L(M) would be recursive and therefore not in r.e. – r.
- **Let M be a TM, and suppose that M loops forever on some string x.**

- Question: Is L(M) recursively enumerable?
  Answer: Yes! By definition it is.

- Question: Is L(M) recursive?
  Answer: Don't know. Although M doesn't always halt, some other TM M' may exist

  such that L(M') = L(M) and M' always halts.

- Question: Is L(M) in r.e. – r?
  Answer: Don't know.

Closure Properties for Recursive and Recursively Enumerable Languages

- **TMs Model General Purpose Computers:**
  - If a TM can do it, so can a GP computer
  - If a GP computer can do it, then so can a TM

  If you want to know if a TM can do X, then some equivalent question are:
  - *Can a general purpose computer do X?*
  - *Can a C/C++/Java/etc. program be written to do X?*

  For example, is a language L recursive?
  - *Can a C/C++/Java/etc. program be written that always halts and accepts L?*

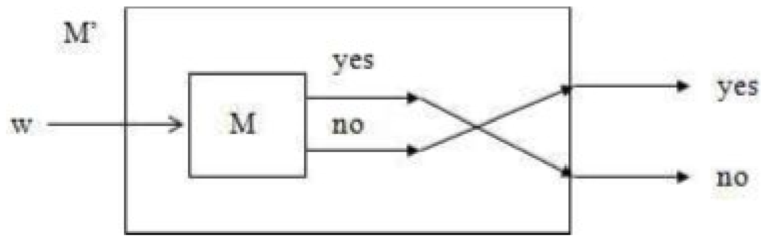
- **TM Block Diagrams:**
  - If L is a recursive language, then a TM M that accepts L and always halts can be pictorially represented by a "chip" that has one input and two outputs.

  - If L is a recursively enumerable language, then a TM M that accepts L can be pictorially represented by a "chip" that has one output.

  - Conceivably, M could be provided with an output for "no," but this output cannot be counted on. Consequently, we simply ignore it.

– **Theorem:** The recursive languages are closed with respect to complementation, i.e., if L is a recursive language, then so is

Proof: Let M be a TM such that L = L(M) and M always halts. Construct TM M' as follows:



- **Note That:**
  - M' accepts iff M does not
  - M' always halts since M always halts

From this it follows that the complement of L is recursive. •

- **Theorem:** The recursive languages are closed with respect to union, i.e., if L_1 and L_2 are recursive languages, then so is

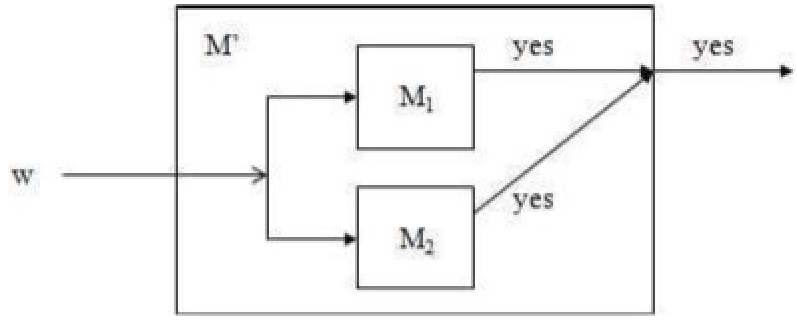
Proof: Let M_1 and M_2 be TMs such that $L_1 = L(M_1)$ and $L_2 = L(M_2)$ and M_1 and M_2 always halts. Construct TM M' as follows:

- **Note That:**
  - $L(M') = L(M_1) \cup L(M_2)$
    - •$L(M')$ is a subset of $L(M_1) \cup L(M_2)$
    - •$L(M_1) \cup L(M_2)$ is a subset of $L(M')$
  - M' always halts since M_1 and M_2 always halt

It follows from this that $L_3 = L_1 \cup L_2$ is recursive.

- **Theorem:** The recursive enumerable languages are closed with respect to union, i.e., if L_1 and L_2 are recursively enumerable languages, then so is $L_3 = L_1 \cup L_2$

Proof: Let M_1 and M_2 be TMs such that $L_1 = L(M_1)$ and $L_2 = L(M_2)$. Construct M' as follows:



- **Note That:**
  - L(M’) = L(M_1) U L(M_2)
    - L(M’) is a subset of L(M_1) U L(M_2)
    - L(M_1) U L(M_2) is a subset of L(M’)
  - M’ halts and accepts iff M_1 or M_2 halts and accepts

It follows from this that   is recursively enumerable.

2. The Halting Problem – Background

- **Definition:** A <u>decision problem</u> is a problem having a yes/no answer (that one presumably wants to solve with a computer). Typically, there is a list of parameters on which the problem is based.
  - Given a list of numbers, is that list sorted?
  - Given a number x, is x even?
  - Given a C program, does that C program contain any syntax errors?
  - Given a TM (or C program), does that TM contain an infinite loop?

From a practical perspective, many decision problems do not seem all that interesting. However, from a theoretical perspective they are for the following two reasons:
  - Decision problems are more convenient/easier to work with when proving complexity results.
  - Non-decision counter-parts are typically at least as difficult to solve.

- **Notes:**
  - The following terms and phrases are analogous:

Algorithm	-	A halting TM program
Decision Problem	-	A language
(un)Decidable	-	(non)Recursive

Statement of the Halting Problem

- **Practical Form:** (P1)
  Input: Program P and input I.
  Question: Does P terminate on input I?

- **Theoretical Form:** (P2)
  Input: Turing machine M with input alphabet Σ and string w in Σ*.
  Question: Does M halt on w?

- **A Related Problem We Will Consider First:** (P3)
  Input: Turing machine M with input alphabet Σ and one final state, and string w in Σ*.
  Question: Is w in L(M)?

- **Analogy:**
  Input: DFA M with input alphabet Σ and string w in Σ*.
  Question: Is w in L(M)?

  Is this problem decidable? Yes!

- **Over-All Approach:**
  - We will show that a language L_d is not recursively enumerable
  - From this it will follow that is not recursive
  - Using this we will show that a language L_u is not recursive
  - From this it will follow that the halting problem is undecidable.

3. The Universal Language

- Define the language L_u as follows:

  L_u = {x | x is in {0, 1}* and x = <M,w> where M is a TM encoding and w is in L(M)}

- Let x be in {0, 1}*. Then either:

1. x doesn't have a TM prefix, in which case x is **not** in L_u
2. x has a TM prefix, i.e., x = <M,w> and either:
   a) w is not in L(M), in which case x is **not** in L_u
   b) w is in L(M), in which case x is in L_u

- **Compare P3 and L_u:**

  (P3):
  Input: Turing machine M with input alphabet Σ and one final state, and string w in Σ*.

- **Notes:**
  - L_u is P3 expressed as a language
  - Asking if L_u is recursive is the same as asking if P3 is decidable.
  - We will show that L_u is not recursive, and from this it will follow that P3 is un-decidable.
  - From this we can further show that the halting problem is un-decidable.
  - Note that L_u is recursive if M is a DFA.

4. Church-Turing Thesis

- There is an effective procedure for solving a problem if and only if there is a TM that halts for all inputs and solves the problem.

- There are many other computing models, but all are equivalent to or subsumed by TMs. *There is no more powerful machine* (Technically cannot be proved).

- DFAs and PDAs do not model all effective procedures or computable functions, but only a subset.

- If something can be "computed" it can be computed by a Turing machine.

- Note that this is called a ***Thesis***, not a theorem.

- It can't be proved, because the term "can be computed" is too vague.

- But it is universally accepted as a true statement.

- Given the ***Church-Turing Thesis***:

  - What does this say about "computability"?

  - Are there things even a Turing machine can't do?

  - If there are, then there are things that simply can't be "computed."

    - Not with a Turing machine

    - Not with your laptop
    - Not with a supercomputer
  - There ARE things that a Turing machine can't do!!!
- The ***Church-Turing Thesis:***
  - In other words, there is no problem for which we can describe an algorithm that can't be done by a Turing machine.

The Universal Turing machine

- If Tm's are so damned powerful, can't we build one that simulates the behavior of any Tm on any tape that it is given?
- Yes. This machine is called the ***Universal Turing machine***.
- How would we build a Universal Turing machine?
  - We place an encoding of any Turing machine on the input tape of the Universal Tm.
  - The tape consists entirely of zeros and ones (and, of course, blanks)
  - Any Tm is represented by zeros and ones, using unary notation for elements and zeros as separators.
- Every Tm instruction consists of four parts, each a represented as a series of **1**'s and separated by **0**'s.
- Instructions are separated by **00**.
- We use unary notation to represent components of an instruction, with
  - 0 = **1**,
  - 1 = **11**,
  - 2 = **111**,

- 3 = **1111**,
- n = **111...111** (n+1 1's).

- We encode q_n as <u>$n + 1$</u> 1's
- We encode symbol a_n as <u>$n + 1$</u> 1's
- We encode move left as 1, and move right as 11

  1111011101111101110100101101101100

  q_3, a_2, q_4, a_2, L q_0, a_1, q_1, a_1, R

- Any Turing machine can be encoded as a unique long string of zeros and ones, beginning with a **1**.
- Let T_n be the Turing machine whose encoding is the number n.

5. Linear Bounded Automata

- A Turing machine that has the length of its tape limited to the length of the input string is called a linear-bounded automaton (LBA).
- A linear bounded automaton is a 7-tuple *nondeterministic* Turing machine M = (Q, S, G, d, q_0,q_{accept}, q_{reject}) except that:

  a. There are two extra tape symbols < and >, which are not elements of G.

  b. The TM begins in the configuration (q_0≤x>), with its tape head scanning the symbol < in cell 0. The > symbol is in the cell immediately to the right of the input string x.

  c. The TM cannot replace < or > with anything else, nor move the tape head left of < or right of >.

Context-Sensitivity

- *Context-sensitive production* any production □ □ □ satisfying |□≤| □|.
- *Context-sensitive grammar* any generative grammar $G = \langle □□, □, \Pi, □ \rangle$ such that every production in Π context-sensitive.
- No empty productions.

Context-Sensitive Language

- Language *L context-sensitive* if there exists context-sensitive grammar *G* such that either $L = L(G)$ or $L = L(G) \cup \{ \}$.

- **Example**:
  The language $L = \{a^n b^n c^n : n \geq 1\}$ is a C.S.L. the grammar is

  S → abc/ aAbc,

  Ab → bA,

  AC → Bbcc,

  bB → Bb,

  aB → aa/ aaA

  The derivation tree of $a^3b^3c^3$ is looking to be as following

  $S \Rightarrow aAbc$

  $\Rightarrow abAc$

  $\Rightarrow abBbcc$

  $\Rightarrow aBbbcc \quad \Rightarrow aaAbbcc$

  $\Rightarrow aabAbcc$

  $\Rightarrow aabbAcc \quad \Rightarrow aabbBbccc$

  $\Rightarrow aabBbbccc$

  $\Rightarrow aaBbbbccc$

  $\Rightarrow aaabbbccc$

CSG = LBA

- A language is accepted by an LBA iff it is generated by a CSG.
- Just like equivalence between CFG and PDA
- Given an x ∈ CSG G, you can intuitively see that and LBA can start with S, and nondeterministically choose all derivations from S and see if they are equal to the input string x. Because CSL's are non-contracting, the LBA only needs to generate derivations of length ≤ |x|. This is because if it generates a derivation longer than |x|, it will never be able to shrink to the size of |x|.