AUTOMATA FORMAL LANGUAGES AND LOGIC

Lecture notes
Undecidability

Prepared by:
Ms. Preet Kanwal
Associate Professor

Department of Computer Science & Engineering
PES UNIVERSITY
(Established under Karnataka Act No.16 of 2013)
100-ft Ring Road, BSK III Stage, Bangalore – 560 085

Table of Contents		
#	Topic Name	Page No.
1.	The Church Turing Thesis	2
2.	The Universal Turing Machine	3-4
3.	The Chomsky Hierarchy ● Recursively Enumerable Sets ● Non-Recursively Enumerable Sets (The Diagonalization Method) ● The Idea of Acceptance & Membership ● Undecidable Problems (Definition & Examples) ● The Complete Chomsky Hierarchy	5 5-7 8-11 12-13 14 15
4.	Undecidable Problems : ● PCP ● Idea of Reduction ● Context Free Languages - Empty Intersection Problem ● Turing Machine Acceptance Problem ● Turing Machine Halting Problem ● Other Example	16 16-20 21 21-22 22-24 24 25

The Church Turing Thesis:

A **computable function** is a function that a Turing Machine can compute.

An **algorithm** is defined simply as a TM that does not enter into an infinite loop for any input.

- It must terminate after a finite number of steps in either in final or non-final state.
- It is a systematic procedure

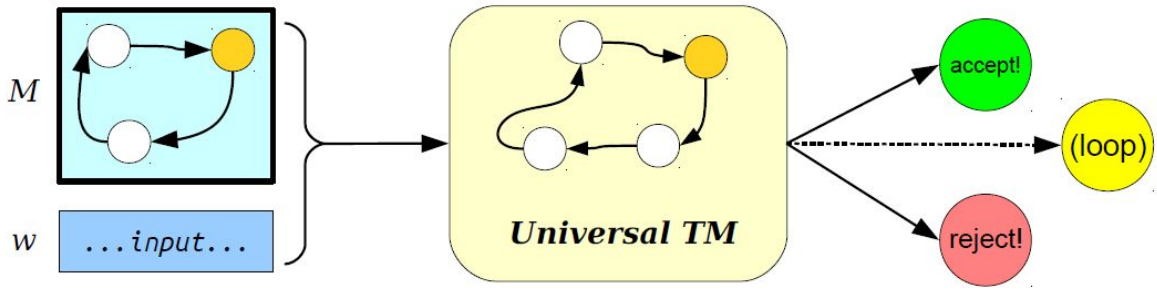
Alonzo Church and Alan Turing showed anything that can be computed can be computed by Turing Machine.

- They independently showed the result (Alonzo church came up with Lambda Calculus)
- This statement has no proof and it's not a theorem. However no one so far has been able to find a computing problem that TM cannot compute.
- We can also say, only functions that are computable by TM are computable.
- Consider the set of all possible Turing Machines ={M1,M2,M3,…} and set of all languages accepted by each Turing Machine = { L(M1), L(M2),…} this set of languages corresponds to the set of all computable functions or we can say, set of all valid computer programs that one could write. This is the largest class of formal languages that are computable and are called Recursively Enumerable Languages.
- From Practical point of view, it is perhaps the most inefficient computing platform one can build.
- But its simplicity is the basis for proving some of the most important theoretical results in computer science
- Multiple attempts have been made to enhance / modify the design of TM to make it more powerful.
- However all the resulting variations turned out to be strictly equivalent to the standard Turing machine in terms of the set of the functions that it can complete thereby supporting Church Turing Thesis.

The Universal Turing Machine

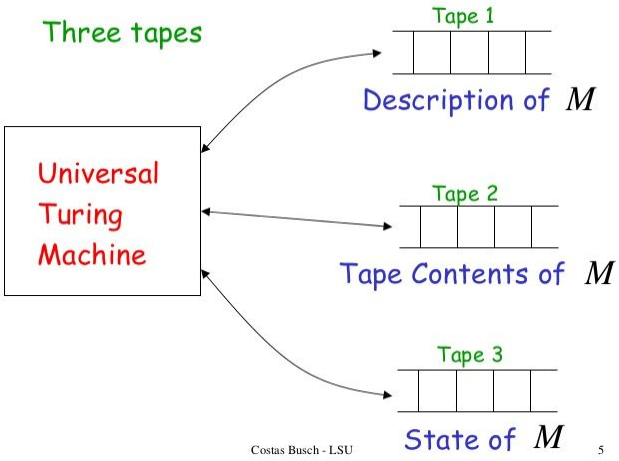
- Until now , we have build TM for each computable function
- The TM which performs addition operation, cannot multiply and vice versa.
- Turing overcame this apparent limitation and showed how a single computing machine can capture / compute any computable function.
- It is called Universal Turing Machine
- A UTM M_u simulate the computation of any TM M on a given input w
- M_u is similar to the compiler software which can compile other programs.

- The observable behavior of U_{TM} is the following:
  - If M accepts w, then U_{TM} accepts $\langle M, w \rangle$.
  - If M rejects w, then U_{TM} rejects $\langle M, w \rangle$.
  - If M loops on w, then U_{TM} loops on $\langle M, w \rangle$.
- **U_{TM} accepts $\langle M, w \rangle$ if and only if M accepts w.**



- M_u is a three tape machine (3 read/write heads)
  - Tape 1: input-output tape
  - Tape 2: Compiled program tape contains encoding of the TM M
  - Tape 3: Working memory tape, contains current state of TM M being simulated

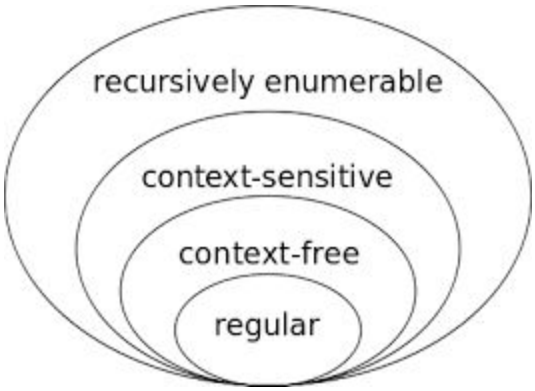
- Working of M_u
  1. Look at current state in working memory and current symbol on input-output tape
  2. With information of current state and current symbol, scan the compiled program tape to find a transition for the given configuration.
  3. Execute the transition i.e. write new symbol on input-output tape and move its read/write head
  4. Repeat above steps until there are no more transitions in compiled program
  5. It halts and accepts or halts and rejects.
- We require a way to encode transition of M as a binary string.



- This idea of Universal TM has been implemented in modern digital computers.
- Programs written in High Level Language are compiled by a compiler.
- Turing's idea of storing an encoded table of instructions (i.e. program) of a TM on one of the tapes of UTM,made it possible for us to build modern computers that can compute practically anything that is computable. Also Turing machines for simple computable functions can be combined to compute more complex functions. This is possible as we can easily concatenate encoded programs of multiple Turing machines to create a program for a complex Turing machine.
- This is similar to what we do today in the name of modularity in programming.

The Chomsky Hierarchy

The big picture of formal languages, grammar and computing machines known as the Chomsky hierarchy. We have already looked at two main classes of formal languages namely, Regular Languages and Context Free Languages. Turing machines are way more powerful than PDA's in terms of solving more problems.



Language of Turing Machine is called Recursively Enumerable Language or Semi decidable or partially decidable or Turing Recognizable.

Why is the language recognized by a Turing Machine called Recursively enumerable?

A set is called Recursively enumerable (RE) if there exists a procedure / algorithm that can enumerate all the members of the set.(This algorithm may run for ever) or we can say, we can write a TM to enumerate strings in such a language or the one in machine we can generate any given element of the set in finite time, for example:

1) **Set of all Natural Numbers is enumerable.**
   We can write a program to enumerate the set of all natural numbers N.
   i.e. we can build a TM for enumerating natural numbers for example by writing each successive number on to its tape
   Such a program will obviously never terminate because the set N is infinite, the machine will not halt. However, give a particular finite number 'n' irrespective of how large 'n' is it will be written in finite time. Hence this set is enumerable.
   Note: Any set which can be placed in one correspondence with the set of natural numbers 'N' (i.e. we can index each element, it makes sense to talk to ith element of the set) is called **countably infinite.**

2) **Set of all even numbers is enumerable.**
   N={1,2,3,4,…}
   E={2,4,6,8,…}
   f(x)=2x

3) **Σ* is enumerable**

Σ*is the set of all strings over the alphabet. It is a countably infinite set , that means we have a way to enumerate strings ∈ **Σ***

Σ*enumerable procedure:

- We would print strings in the language L=**Σ***in alphabetical order.(lexicographic order)
  - For example if **Σ**= {a,b,c,}
  - As per our enumeration procedure, we print { **λ, a, aa, aaa, aaaa, …}**
  - we will never be able to print strings that begin with b or c
  -
- Hence we must print the strings in proper order i.e. by order of increasing length.
  - {**λ, a, b, c, aa, ab, ac, ba, bb, bc, ca, cb, cc, aaa, aab, ...}**
- If we can enumerate any such set , we can immediately talk about ith element of the set and thereby map the elements to the set of natural numbers.
- In this case , given any string of finite length , w ∈ **Σ***, we will be able to print w in finite time following our enumeration procedure.

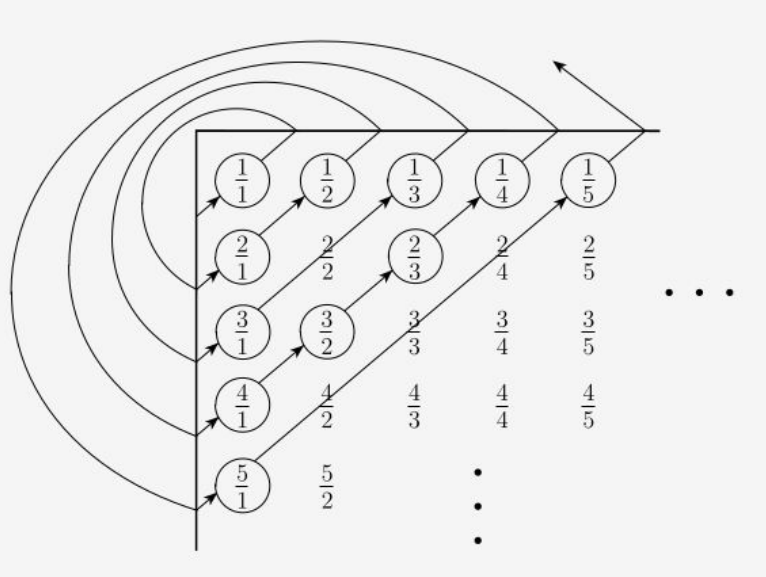
4) Set of all Turing Machines are enumerable

- We know from our study of UTM that a TM can be encoded as a binary string
- Binary alphabet **Σ** ={0,1}
  - Set of all binary strings is **Σ***i.e. (0+1)*
  - **Σ*** is an infinite set
- we must note that , not all binary strings are valid encodings of TMs
  - With this we can conclude that set of Turing Machines is proper subset of **Σ***
- Also, TM always has a finite description as there are a finite number of transitions and tape symbols.
- Since we have a way to enumerate all strings in **Σ*.** We can enumerate all TM's. Since every Turing Machine can be encoded as a finite string over a finite alphabet, the set of all TM's is enumerable.

5) Q={m/n | m,n ∈ N} is a set of positive rational numbers . Q is countable

Proof:

We provide a correspondence with N. we can list all members and elements of Q. we can make an infinite matrix containing all positive rational numbers.



Row headings -> indicates numerator

Column headings -> indicates denominator

With Row(i) column(j) we get, rational numbers i/j

This matrix can be turned into a list. List of all elements on the diagonals starting from the corner. First diagonal contains a single element 1/1. Second diagonal contains 2/1, 1/2 . Third diagonal, a complication arises , it contains 3/1, 2/2, 1/3 , we see 1/1 = 2/2, we must skip duplicate elements.

Hence our list will have: 1/1, 2/1,1/2 , 3/1, 1/3, …

- All the above sets are infinite and have the same size (cardinality) as that of a set of natural numbers as we are able to keep the elements from both the sets in one to one correspondence.
- Cantor proposed this idea to compare or measure the size of infinite sets.
- Cantor's idea of counting the elements, pairs with the elements of the other sets.
- Note: A set is countable if either it is finite or it has same size as N

Non-Recursively Enumerable Sets

- Do all infinite sets have the same size / cardinality?
- Are there sets that are not enumerable (cannot be kept in one to one correspondence with sets of Natural numbers?
- Is it same as saying: Are there sets for which no TM enumerator can be constructed?
    - The answer is yes. Such sets are called Non-Recursively Enumerable.

1) Set of real numbers R is not enumerable.

- Consider the set of real numbers. Try enumerating all elements between 1 and 2, 1.0, 1.00, 1.000, ... , 1.1, 1.10, 1.100…
- We face trouble listing the elements unless we place a limit on the precision of the numbers. If the precision is limited, we can enumerate all real numbers. We can write the numbers ordered by length first and then by value.
- Example: 0.1, 1.0, 0.2, 0.3, … 0.9, 1.0, … 9.9, 0.01, 0.02, …
- Print all the numbers that have one decimal place only, then the numbers within two decimal places and so on… But if precision is unlimited we cannot ever enumerate a real number in finite time in any attempt.
- Hence such a set is not enumerable,

- There doesn't exist any algorithm to enumerate all elements of the set
- It doesn't make sense to talk about i^{th} real number
- There are infinitely many elements between two elements of the set
- We cannot enumerate any element of the set infinite time
    - Such sets are also called as uncountably infinite, such sets have "too many elements"
- Formal proof to show to show that set R is not enumerable:
    - Proof uses the technique called **Cantor's Diagonalization**
    - It is a proof by contradiction
    - **Our Aim**: our aim is to set up a contradiction to prove that our assumption is wrong.
    - **Assumption**: Set R is enumerable
    - Hence we enumerate all elements of R, let's say

n	$f(n)$
1	3.14159...
2	55.55555...
3	0.12345...
4	0.50000...
⋮	⋮

- This table has an infinite number of rows and columns
- However we can talk about 1st real number , 2nd real number and so on
- Consider the left to right diagonal which contains the 1st decimal place of first real number (1 in our case) 2nd decimal place of 2nd real number (2 in our case) and so on . If we read this diagonal as a real number, we get
  R_d = .12275

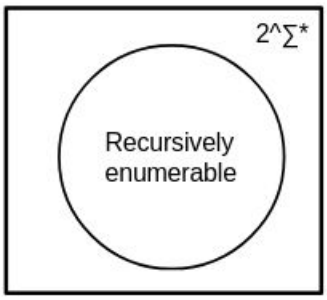
- Let us insert these element (you could use any procedure may add a 2 , subtract 1, add a3, add a 4 etc..) basically choose some digit between 1 and 8) in our case we add a 1 we get,
- R_d' = .23386. Result is also a real number.
- Question is, Does this inverted diagonal number R_b' exist in the enumerated list? i.e. is it the same as one of the rows in the table we have constructed? we see that it is not same as first row since it differs in first decimal place, it's not same as second number as it differs in 2nd decimal place, it is not same as ith row, as it differs in ith decimal place. It differs from every enumerated real number in the corresponding diagonal element (i.e. ith decimal place). But, we stated with the assumption that the real number in set R has been enumerated in the table. Yet we got a new number. This is a contradiction, hence our assumption is wrong.
- We have proved our assumption is wrong using the diagonalization method. Hence, Set R is not enumerable

2) Set of all languages are not enumerable

- We can use the same method of diagonalization to prove that sets of all languages are not enumerable. Any formal language L is a subset of **Σ*** **L⊆Σ* , we** must remember that **Σ* is** enumerable(countably infinite)
- However, a set of formal languages is not enumerable (uncountably infinite). Since any language is a subset of **Σ* .**
- Total number of subsets that that can be formed using the set **Σ* is nothing but the power set of Σ* = 2^Σ***.
- Formal Proof: set of all languages over Σ are enumerable. This means the languages can be numbered as L1, L2, L3, …
- Let's construct a 2-D matrix with an infinite number of rows .
- Rows-> are enumerated as languages from the power set **2^Σ***.(each row represents a language).we can also see the rows as enumeration of Turing Machines with row L_i corresponding to the language of the ith Turing Machine.
- Columns-> Columns in the matrix are all the strings from **Σ*={s1,s2,s3, …} (we know that Σ* is enumerable)**

	S1	S2	S3	S4.....
L1	1	0	1	1
L2	1	1	0	1
L3	0	1	0	0
L4	1	0	0	0
.				
.				
L15	1	1	0	0

- 1 indicates string $s_i \in L_j$, indicates string $s_i \notin \mathbf{L_j}$,
  - **L1={s1,s3,s4…**
  - **L2={s1,s2,s3…**
  - **L3={s2…**
  - **L4={s1,…**
  - **L15={s1,s2,…**
- Consider left to right diagonal, this will represent another language lets say Ldiag
- Thus, L_{diag}={s1,s2,…}
- This language L_{diag} must be one of the enumerated languages(must be same as one of the rows)In this case L_{diag} = L15
- Thus the language is computable and recursively enumerable since we can construct Turing machine for such a language(as per our assumption)
- Complement this language, what we get is a new language $L_{non\_RE} = L_{diag}'$
- $L_{non\ RE}$={ s3,s4,…}
- This new language is not the same as any of the enumerated elements (rows)because it differs from ith diagonal place. Hence it is a new language and not the same as any of the enumerated languages.
- This is contradiction; hence our assumption that all languages are enumerable is wrong
- Note: From our previous study of universal Turing Machine, we know that we can encode all the transitions of a Turing Machine as a binary string. Since we can enumerate all possible binary strings (i.e. Σ* where Σ={0.,1}). Hence, we can enumerate all Turing Machines as each Turing Machine has a finite description and can be encoded as a unique binary string. The total number of possible Turing Machines is less than size of Σ* for the binary alphabet. So we can just enumerate all possible strings and discard invalid encodings ace to the sentence used for encoding.
- There are countably infinite numbers of Turing Machines in the world. But there are uncountably infinite number of formal languages(we just proved)
- There exists a formal language for which we cannot construct a TM i.e. there exists a language which is not Recursively Enumerable.
- Set of Recursively Enumerable Languages is a proper subset of a set of all formal languages.

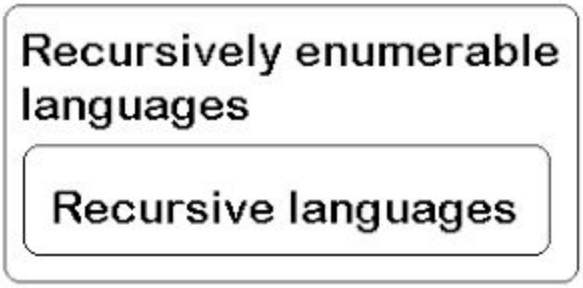


- Not Recursively Enumerable means not computable. Most languages are not Recursively Enumerable but that's okay, as these innumerable, uncountable languages are not interesting. Not computable, most of the times these languages have no direct description. Because if we could describe directly, we could convert such a description to an algorithm and there by a Turing machine.

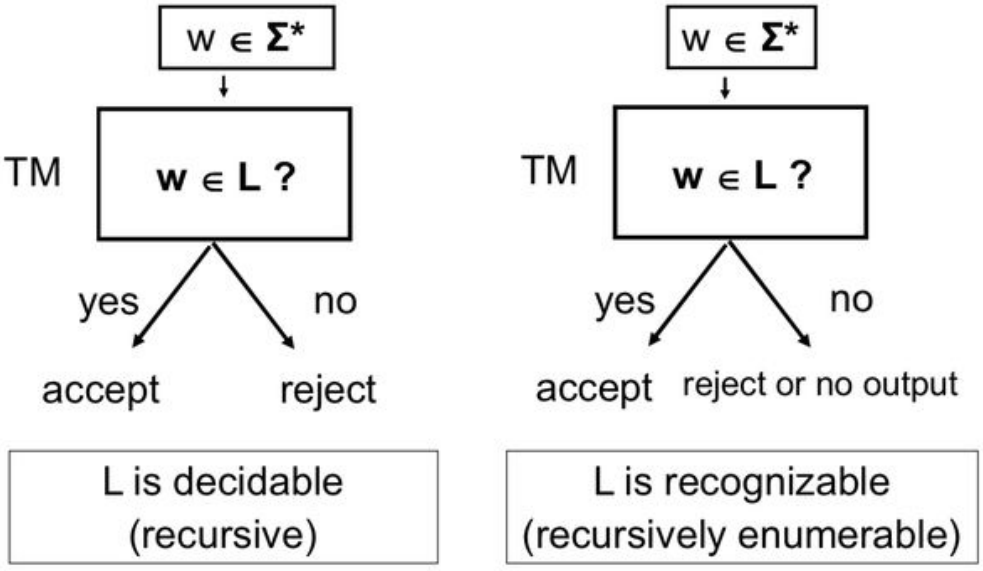
The Idea of Acceptance and Membership

We know that languages accepted by TM are called RE Languages.

- If a Turing Machine halts on every input it is called TM decider. Language of TM Decider is Recursive language. Such TM's complete membership of strings in the language i.e.
    - For any given string w $\in \Sigma^*$.
    - Turing machine decider halts and accept if w $\in$ **L(M)**, or halts and reject if w $\in$ **L(M)**
    - If the language has a membership function its complement also has a membership function. Such languages are called recursive languages or Turing decidable languages. Recursive languages are a proper subset of recursively enumerable languages.

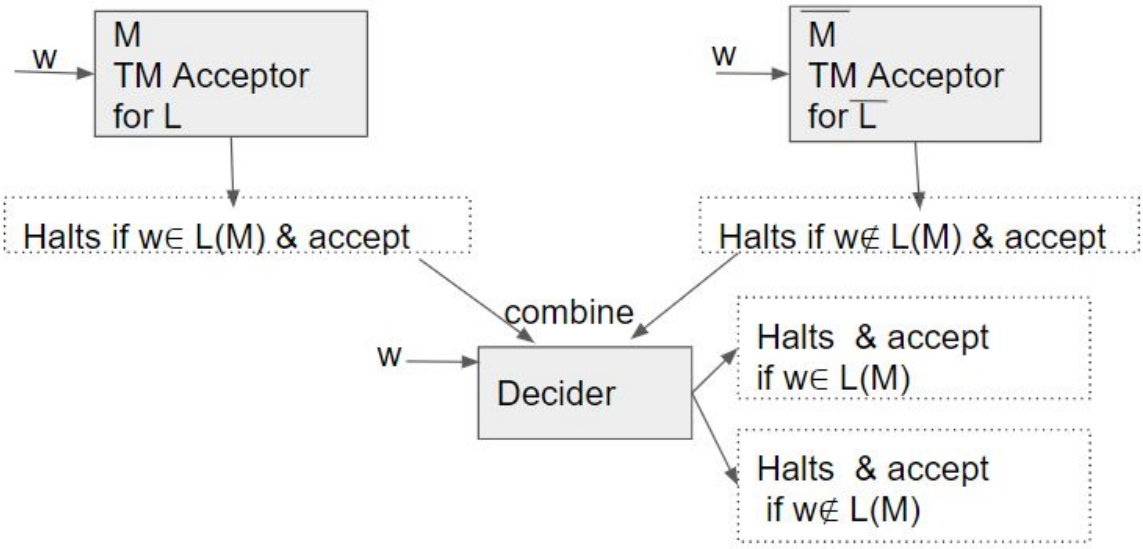


- There exist RE languages (for example L_{diag}, PCP) which are not recursive i.e. we can create only the acceptor for such languages. i.e.



TM created is called TM acceptor. Such machines do not implement membership function. Hence languages like L_{diag} or PCP have an Acceptor but not Decider and are called undecidable.

- RE Languages are not closed under complement because if they were, we could have an acceptor for L and its complement L', the language will become recursive.



Note: Set of languages where both L and L' are RE are called Recursive languages.

Summary:

RE languages

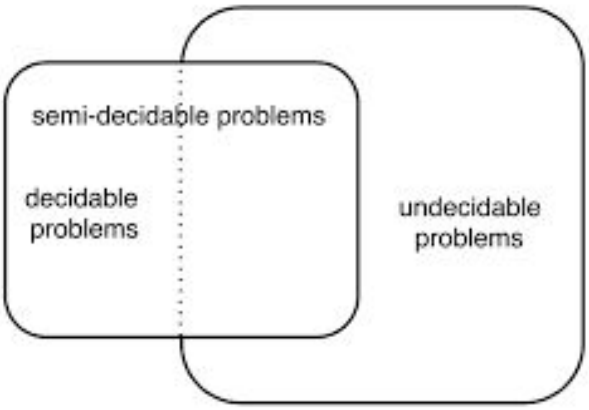
- Have enumeration procedure
- No membership function
- TM acceptor can be implemented

Recursive Languages:

- Have Enumeration Procedure
- Membership function
- Proper subset of RE languages
- TM decider is implemented

Undecidable Problems (No algorithm)

Definition :



A language L is **undecidable** if there is no TM M that halts on every input.

- **Either L is Non-RE**

  i.e. No Turing machine can be constructed for such a language , no enumeration procedure to enumerate all elements of such a language set

- **L is RE but not decidable.**

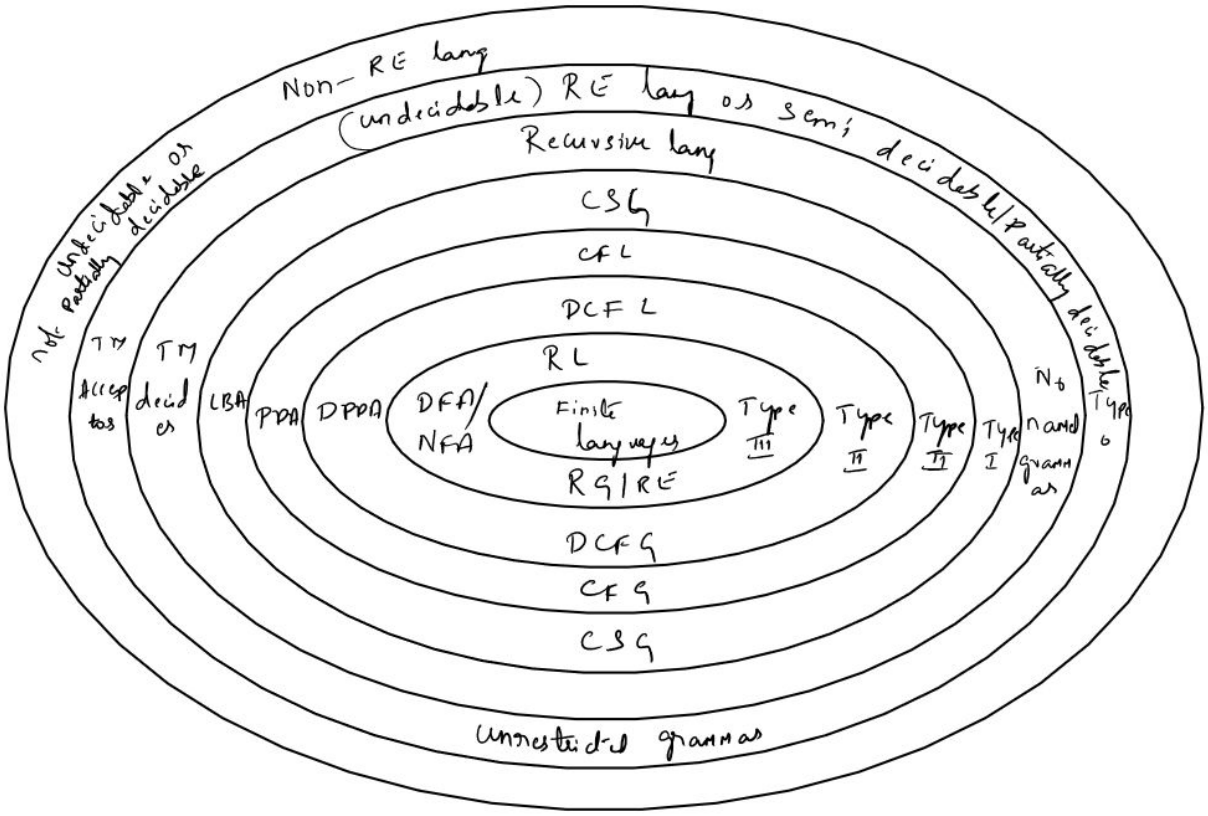
Example of Undecidable problems:

- **Recursively Enumerable but not decidable**
  a) PCP
  b) L1 ∩ L2
  c) Halting Problem
  d) Acceptance problem of TM

- **Non Recursively Enumerable**
  a) Set of real numbers
  b) Set of all languages ∈ **2**^Σ*

The Complete Chomsky Hierarchy:

Summary of relationship between class of formal languages, machines and grammars, such a hierarchy of languages is known as Chomsky Hierarchy in honor of Noam Chomsky who developed much of theory of formal languages and grammars.



Undecidability or unsolvability:

We begin to investigate the power of algorithms to solve problems. We demonstrate certain problems that can be solved algorithmically and others that cannot. Our objective is to explore the limits of algorithmic solvability.

Why should one study unsolvability?

First, knowing when a problem is algorithmically unsolvable is useful because then you realize that the problem must be simplified or altered before you can find an algorithmic solution. Like any tool, computers have capabilities and limitations that must be appreciated if they are to be used well.

The second reason is cultural. Even if you deal with problems that clearly are solvable, a glimpse of the unsolvable can stimulate your imagination and help you gain an important perspective on computation.

In one type of unsolvable problem, you are given a computer program and a precise specification of what that program is supposed to do (e.g., sort a list of numbers). You need to verify that the program performs as specified (i.e.,that it is correct). Because both the program and the specification are mathematically precise objects, you hope to automate the process of verification by feeding these objects into a suitably programmed computer. However, you will be disappointed. The general problem of software verification is not solvable by computer.

For any undecidable language, either it or its complement is not Turing-recognizable.

Post Correspondence Problem(PCP)

- Undecidable decision problem (RE and undecidable)
- Introduced by emil post in 1946

Introduction :
Somebody gives you two sets of strings

Example 1:

	List A	List B
1	10	10110
2	111	01
3	110110	1010

[In general,any number of pairs of strings can be given -here we have 3 pairs]

The question is ,can i make a sequence of these numbers for example, 1 1 3 2 2 1

So that when I go ahead and concatenate all the strings from each list A,B in that order (in that sequence) the two strings will be the same.

Sequence	1	1	3	2	2	1
A	10	10	110110	111	111	10
B	10110	10110	1010	01	01	10110

The question is after concatenating the string from A and B in the given sequence.
Are the resulting strings the same?
Is there a sequence that lets them be the same.
So the post correspondence problem is:
Given two sets of strings A and B(over same □) each having n strings.

A	B
a1	b1
a2	b2
a3	b3
a4	b4 . . . .
an	bn

The problem is to come up with a sequence in which when one concatenates all strings in A(in that order) and concatenates all strings in B (in same order) .The resulting string is the same.

- Is there some order of concatenation that will give id identical results for both the sets.
- Can we find a permutation(arrangement) thar results in sme string.
- If we do not allow repetition of strings since our n is finite(n=3) in our example.We can enumerate all permutations and try each resulting concatenation.
- Hence ,the problem can be solved easily.

If repetition of strings is allowed :

- There will be no limit on the length of the concatenation although n-finite.
- This is the general case of PCP-more complex.
- It turns out that only some instances of PCP have solutions.
- Other pairs of sets A and B ,for which there is no solution,we cannot solve them-may enter into an infinite loop.

There is no algorithm that can solve all instances of PCP.

There will never be an algorithm that can solve all instances of this problem .
Hence,its an example of a computing problem which is unsolvable.
PCP-There are various variations of this problem which are unsolvable ,undecidable.

Example 2:

	List A	List B
1	10	101
2	011	11
3	101	011

Only way to match is strat with 1.

Sequence A	**1**		**3**			**3**			
Sequence A	1	0	1	0	1	1	0	1	
Sequence B	**1**			**3**			**3**		
Sequence B	1	0	1	0	1	1	0	1	1 (extra 1)

You can only continue with 3.The only way to match then is to continue with 3.
Sequence: 1 3 3 3 3 ……..
And none of these ever even out because always the bottom is going to be one bit longer than the top.

Example 3:

	List A	List B
1	bb	bbb
2	baa	aab
3	bbb	bb

Sequence A	**1**		**3**			**3**		
Sequence A	b	b	b	a	a	b	b	b
Sequence B	**1**			**2**			**3**	
Sequence B	b	b	b	a	a	b	b	b

Example 4:

	List A	List B
1	a	baa
2	ab	aa
3	bba	bb

Sequence A	**3**			**2**		**3**			**1**
Sequence A	b	b	a	a	b	b	b	a	a
Sequence B	**3**		**2**		**2**		**1**		
Sequence B	b	b	a	a	b	b	b	a	a

Example 5:

	List A	List B
1	a	aa
2	aa	ab
3	b	bb

Sequence A	**1**	**2**		**3**	**3**			
Sequence A	a	a	a	b	b			
Sequence B	**1**		**2**		**3**		**3**	
Sequence B	a	a	a	b	b	b	b	b

Always 2 b's(bb) extra. No solution to this instance of PCP.

Sequence A	**3**	**3**	**3**			
Sequence A	b	b	b			
Sequence B	**3**		**3**		**3**	
Sequence B	b	b	b	b	b	b

No solution.

Usage of PCP

Once we know PCP is undecidable.We can prove that other things out there are also undecidable.

Idea of Reduction

Given,problem P1 is unsolvable(undecidable).
We reduce P1 to another problem P2 and show that,
Solution of P2 implies solution to P1 i.e if P2 becomes solvable then P1 also becomes solvable.
But ,as we know ,P1is unsolvable ,hence ,P2 is also unsolvable.

Example:

Find out whether two CFL's have something in common or not.

i.e L1 ∩L2=⌀?

- Basically,we reduce PCP to empty intersection.
- We show that ,if there's a method to solve the empty intersection problem ,we can solve PCP too.

How are we going to connect the PCP solution to the empty intersection problem ?

- Take the PCP problem.
- Come up with two grammars(corresponding to two lists)
- Give these grammars to the person who claims to have an hypothetical algorithm to solve the empty intersection problem .
- The person then comes back with the string that both the languages have in common.
- But,if there is no string in common then there is no way to solve this problem.

We took PCP and hid it inside two grammars. Now a person who knows how to check empty intersections between CFL's can solve PCP.

Reduction:
Reduction is the method of changing the input of hard problem (PCP) to the input of an unknown problem (empty n) and showing that the answer to unknown problem is true only if the answer to PCP is true.
Hence, we can't do an empty intersection as it implies to PCP and a solution to PCP is impossible.

Example 1:
PCP Problem

	List A	List B
1	a	baa
2	ab	aa
3	bba	bb

Grammar 1
$S_A \rightarrow 1S_{Aa}a|10111S_Ab|10S_Ac|-$

Grammar 2

$S_B \rightarrow 111S_Ba|10S_Bb|0S_Bc|-$

a's,b's,c',s -keep track of the sequence (way if remembering sequences).

We know this PCP problem has a solution : 2113

Lets generate the string 101111110

$S_A \Rightarrow 10111S_Ab$

$\Rightarrow 101111S_Aab$

$\Rightarrow 1011111S_Aaab$

$\Rightarrow 101111110S_Acaab$

$\Rightarrow 101111110\text{-}caab$

caab=3112 (reverse as its nested)

We can generate the same string using S_B.

- Since we have a's,b's ,c's we can be sure that strings are formed using the same sequence(makes sure we have a solution for PCP).
- Hence if S_A and S_B generate the same string,there is a solution to PCP.
- We actually try to solve PCP using L1∩ L2=∅ as a subroutine .As there is no way to solve PCP in the subroutine L1∩ L2=∅ cannot exist.

2. A_{TM} = {⟨M, w⟩| M is a TM and M accepts w} is undecidable

Note : Recursively Enumerable but not Decidable

the problem of determining whether a Turing machine accepts a given input.

Proof using Diagonalization :

- Assume that a TM H decides A_{TM} .
- We construct a new Turing machine D with H as a subroutine.
- Finally, run D on itself.
- Thus, the machines take the following actions, with the last line being the contradiction.
    - H accepts ⟨M, w⟩ exactly when M accepts w.
    - D rejects ⟨M ⟩ exactly when M accepts ⟨M ⟩.
    - D rejects ⟨D⟩ exactly when D accepts ⟨D⟩ ???

we list all TM s down the rows, M 1 , M 2 , . . . , and all their descriptions across the columns, ⟨M 1 ⟩, ⟨M 2 ⟩, ... The entries tell whether the machine in a given row accepts the input in a given column. The entry is accept if the machine accepts the input but is blank if it rejects or loops on that input.

	$\langle M_1 \rangle$	$\langle M_2 \rangle$	$\langle M_3 \rangle$	$\langle M_4 \rangle$	$\cdots$
M_1	*accept*		*accept*		
M_2	*accept*	*accept*	*accept*	*accept*	
M_3					$\cdots$
M_4	*accept*	*accept*			
$\vdots$			$\vdots$		

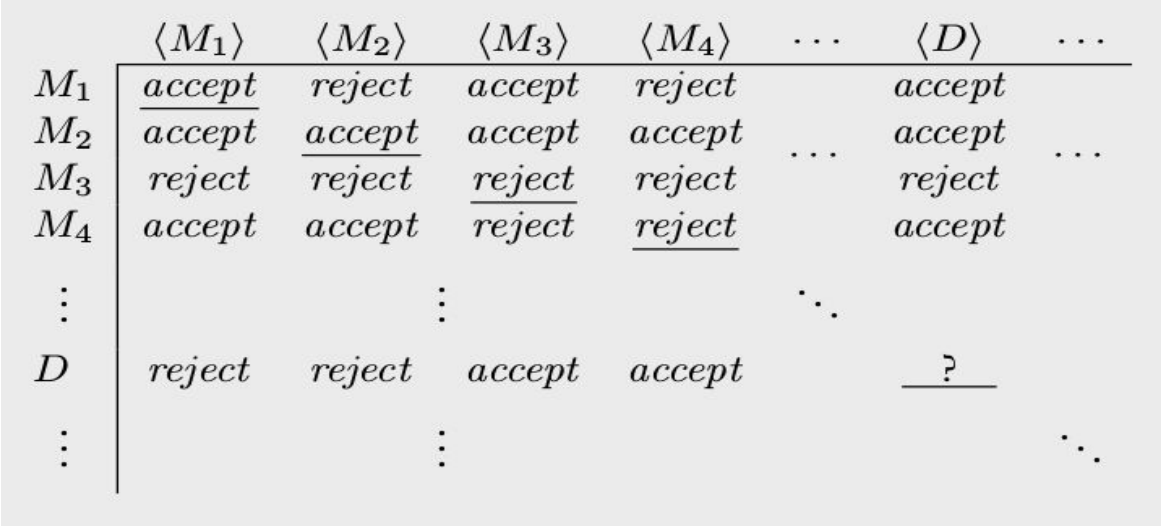
If we run H on inputs corresponding to above table, H rejects input ⟨M 3 , ⟨M 2 ⟩⟩.

H accepts input ⟨M 4 , ⟨M 1 ⟩⟩. Hence, table for H becomes :

	$\langle M_1 \rangle$	$\langle M_2 \rangle$	$\langle M_3 \rangle$	$\langle M_4 \rangle$	$\cdots$
M_1	*accept*	*reject*	*accept*	*reject*	
M_2	*accept*	*accept*	*accept*	*accept*	$\cdots$
M_3	*reject*	*reject*	*reject*	*reject*	
M_4	*accept*	*accept*	*reject*	*reject*	
$\vdots$			$\vdots$		

By our assumption, H is a TM and so is D. Therefore, it must occur on the list M 1 , M 2 , . . . of all TM s.

Note that D computes the opposite of the diagonal entries. The contradiction occurs at the point of the question mark where the entry must be the opposite of itself.



	$\langle M_1 \rangle$	$\langle M_2 \rangle$	$\langle M_3 \rangle$	$\langle M_4 \rangle$	$\cdots$	$\langle D \rangle$	$\cdots$
M_1	*accept*	*reject*	*accept*	*reject*		*accept*	
M_2	*accept*	*accept*	*accept*	*accept*	$\cdots$	*accept*	$\cdots$
M_3	*reject*	*reject*	*reject*	*reject*		*reject*	
M_4	*accept*	*accept*	*reject*	*reject*		*accept*	
$\vdots$		$\vdots$			$\ddots$		
D	*reject*	*reject*	*accept*	*accept*		?	
$\vdots$		$\vdots$					$\ddots$

No matter what D does, it is forced to do the opposite, which is obviously a contradiction. Thus, neither TM D nor TM H can exist.

(Trying to construct Machine for $\mathbf{L_D}$ by using Machine for $\mathbf{A_{TM}}$ as subroutine)

3. Let's consider a related problem, HALT $_{TM}$, the problem of determining whether a Turing machine halts (by accepting or rejecting) on a given input. This problem is widely known as the halting problem. We use the undecidability of A_{TM} to prove the undecidability

of the halting problem by reducing A_{TM} to HALT $_{TM}$. Let

HALT $_{TM}$ = {⟨M, w⟩| M is a TM and M halts on input w}.

Let's assume for the purpose of obtaining a contradiction that TM R decides HALT $_{TM}$. We construct TM S to decide A_{TM} , with S operating as

follows.

S = "On input ⟨M, w⟩, an encoding of a TM M and a string w:

1. Run TM R on input ⟨M, w⟩.

2. If R rejects, reject . If R indicates that M doesn't halt on w, reject because ⟨M, w⟩ isn't in A_{TM} .

3. If R accepts, simulate M on w until it halts.

4. If M has accepted, accept ; if M has rejected, reject ."

Clearly, if R decides $HALT_{TM}$, then S decides A_{TM} . Because A_{TM} is undecidable, $HALT_{TM}$ also must be undecidable.

Reduced A_{TM} to $HALT_{TM}$

(Trying to construct Machine for $\mathbf{A_{TM}}$ by using Machine for $\mathbf{HALT_{TM}}$ as subroutine)

Other Example

Is the given language Recursive ??

L={<M> | M is a TM and there exists a input on machine M halts in less than |<M>| steps}

Solution:

Run M on all inputs of length almost | <M>| , since we are bounding the number of steps that M runs on an input there is no point looking at any string longer than the bound (i.e. |<M>|)

- No possible inputs is finite and as per question number of steps M runs on every input is finite
- M is guaranteed to halt and decide the language.