Session 14

- Non-Recursively Enumerable Languages
- An Undecidable Problem That is RE

Non-Recursively Enumerable Languages

Enumerating the Binary Strings

We assign integers to all the binary strings so that that each string corresponds to one integer, and each integer corresponds to one string.

Here is an approach:

1. Strings are ordered by length,
2. Strings of equal length are ordered lexicographically.

That is, we have

$$\begin{array}{cccccccccccccc} 1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 & 9 & 10 & 11 & 12 & 13 & \\ \updownarrow & \updownarrow & \updownarrow & \updownarrow & \updownarrow & \updownarrow & \updownarrow & \updownarrow & \updownarrow & \updownarrow & \updownarrow & \updownarrow & \updownarrow & \cdots \\ \epsilon & 0 & 1 & 00 & 01 & 10 & 11 & 000 & 001 & 010 & 011 & 100 & 101 & \end{array}$$

$$\begin{array}{cccccccccccccc} 1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 & 9 & 10 & 11 & 12 & 13 & \\ \updownarrow & \updownarrow & \updownarrow & \updownarrow & \updownarrow & \updownarrow & \updownarrow & \updownarrow & \updownarrow & \updownarrow & \updownarrow & \updownarrow & \updownarrow & \cdots \\ 1\epsilon & 10 & 11 & 100 & 101 & 110 & 111 & 1000 & 1001 & 1010 & 1011 & 1100 & 1101 & \end{array}$$

If a binary string w is the ith string, then $1w$ represents the binary integer i.

e.g. The 5337th string is 010011011001, since $1010011011001 = 5337_2$.

Codes for Turing Machines

Now we devise a binary code for Turing machines so that each TM with input alphabet $\{0, 1\}$ may be thought of as a binary string.

Since we just saw how to enumerate the binary strings, we shall then have an identification of the Turing machines with the integers, and we can talk about "the ith Turing Machine, M_i".

Now we represent TM $M = (Q, \{0, 1\}, \Gamma, \delta, q_1, B, \{q_2\})$ as a binary string.

We first assign integers to the states, tape symbols, and directions as follows:

- Assume the states are $q_1, q_2, q_3 \cdots q_k$ for some k. The start state is always q_1, and q_2 is the only accepting state.
- Assume the tape symbols are $X_1, X_2, X_3, X_4 \cdots X_m$ for some m. X_1 is always 0, X_2 is 1, and X_3 is B, the blank. Other tape symbols can be assigned to the remaining integers arbitrarily.
- Refer to direction L as D_1 and direction R as D_2.

Now we encode the transition function δ. Suppose one transition rule is

$$\delta(q_i, X_j) = (q_k, X_l, D_m)$$

for some integers i, j, k, l, m. We shall code the rule by the string

$$0^i 1 0^j 1 0^k 1 0^l 1 0^m$$

A code for entire TM M is:

$$C_1 11 C_2 11 \cdots C_{n-1} 11 C_n$$

where each of the C's is the code for one transition of M.

Example Let $M = (\{q_1, q_2, q_3\}, \{0, 1\}, \{0, 1, B\}, \delta, q_1, B, \{q_2\})$ where δ consists of the rules:

$$\delta(q_1, 1) = (q_3, 0, R), \quad \delta(q_3, 0) = (q_1, 1, R), \quad \delta(q_3, 1) = (q_2, 0, R), \quad \delta(q_3, B) = (q_3, 1, L)$$

The codes for each of these rules, respectively, are

0100100010100 0001010100100 00010010010100 0001000100010010

A code for M is

01001000101001100010101001001100010010010100110001000100010010

Note that

- There are many possible codes for a TM M. In particular, the codes for the s transitions may be listed in any of $s!$ orders, giving $s!$ codes for M.
- Many binary strings do not correspond to any TM at all. For instance, 11001 does not begin with 0, and 00101110100100 has three consecutive 1's.

  If w_i is not a valid TM code, we shall take M_i to be the TM with one state and no transitions. Thus, $L(M_i) = \emptyset$.

In fact, $L(M_i) = \emptyset$ for $i = 1, 2, \cdots, 100$.

The Diagonalization Language

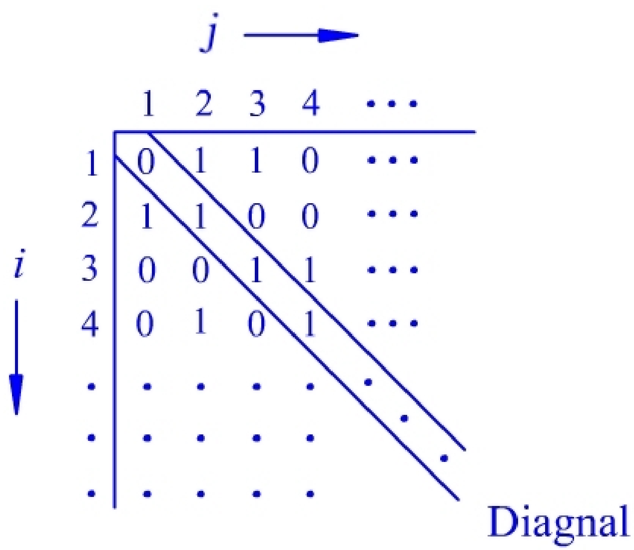
Now, we can make a vital definition

- The language L_d, the diagonalization language, is the set of strings w_i (the ith binary string) such that w_i is not in $L(M_i)$ (the ith Turing machine).

That is, L_d consists of all strings w such that the TM M whose code is w does not accept when given w as input.

Clearly, $\{\epsilon, 0, 1, 00, 10, 11, 000, \cdots, 0000\} \subset L_d$.

The reason L_d is called a “diagonalization” language can be seen if we consider following figure.



This table tells for all i and j, whether the TM M_i accepts input string w_j; 1 means "yes it does" and 0 means "no it doesn't".

We may think of the ith row as the characteristic vector for the language $L(M_i)$; that is, the 1's in this row indicate the strings that are members of this language.

The diagonal values tell whether M_i accepts w_i.

Question: *Can L_d be accepted by some Turing machine?*

Theorem 9.1 *There is no Turing machine that accepts L_d. That is, L_d is not a recursively enumerable language.*

Proof Suppose L_d were $L(M)$ for some TM M. There is at least one code for M, say i, that is, $M = M_i$. Now, ask if w_i is in L_d.

- If $w_i \in L_d = L(M_i)$, then M_i accepts w_i. But by definition of L_d, $w_i \notin L_d$.
- If $w_i \notin L_d = L(M)$, then M_i does not accept w_i. But by definition of L_d, $w_i \in L_d$.

There is a contradiction of our assumption that M exists. ◀


Break for 15 minutes

An Undecidable Problem That is RE

Recursive Languages

We have seen that the diagonalization language L_d is not the recursively enumerable language, that is, that has no Turing machine to accept it.

Now we refine the structure of the recursively enumerable (RE) language into two classes.

1. Languages that are recognized by the TM which always halt eventually.
2. Languages that are not accepted by any TM with the guarantee of halting.

The first class of languages, which corresponds to what we commonly think of as an *algorithm*, has a TM that not only recognizes the language, but it tells us when it has decided the input string is not in the language.

The second class of languages are accepted in an inconvenient way: if the input is in the language, we'll eventually know that, but if the input is not in the language, then the Turing machine may *run forever*, and we shall never be sure the input won't be accepted eventually.

We call a language L recursive if $L = L(M)$ for some Turing machine M such that:

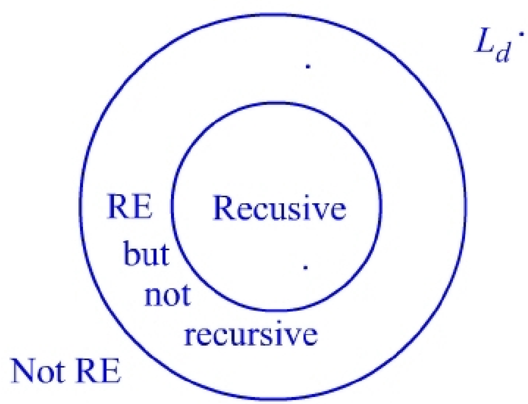
1. If $w \in L$, then M accepts (and therefore halts).
2. If $w \notin L$, then M eventually halts, although it never enters an accepting state.

A TM of this type corresponds to our informal notion of an "*algorithm*", a well-defined sequence of steps that always finishes and produces an answer.

If we think of the language L as a "problem", then problem L is called decidable if it is a recursive language, and it is called undecidable if it is not a recursive language.

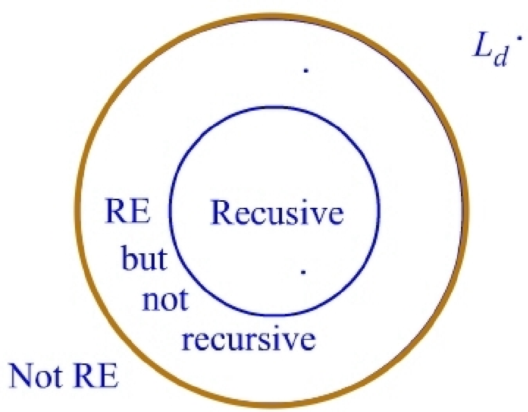
The existence or nonexistence of an algorithm to solve a problem is often of more importance than the existence of some TM to solve the problem.

Thus, dividing problems or languages between the decidable and the undecidable is often more important than the division between the RE and non-RE.



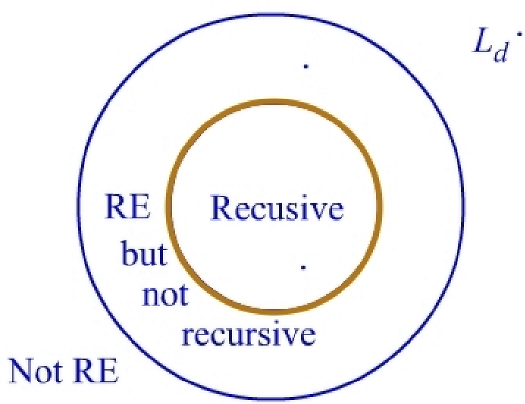
The existence or nonexistence of an algorithm to solve a problem is often of more importance than the existence of some TM to solve the problem.

Thus, dividing problems or languages between the decidable and the undecidable is often more important than the division between the RE and non-RE.



The existence or nonexistence of an algorithm to solve a problem is often of more importance than the existence of some TM to solve the problem.

Thus, dividing problems or languages between the decidable and the undecidable is often more important than the division between the RE and non-RE.

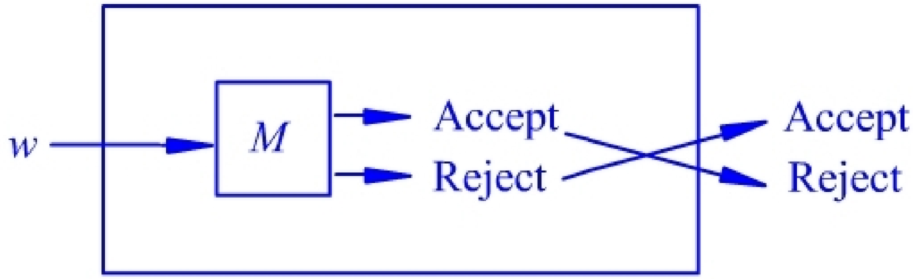


Complements of Recursive Languages

A powerful tool in proving a language to be RE, but not recursive is consideration of the *complement* of the language.

Theorem 9.2 *If L is a recursive language, so is $\overline{L}$. That is, the recursive languages are closed under complementation.*

Proof Let $L = L(M)$ for some TM M that always halts. We construct a TM $\overline{M}$ such that $\overline{L} = L(\overline{M})$.

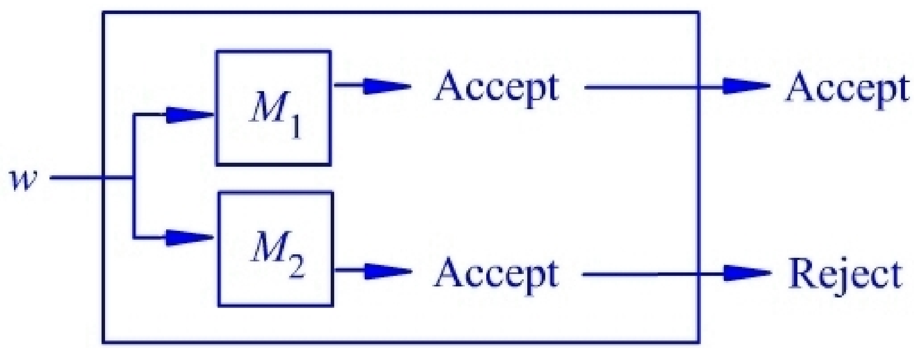


Since M is guaranteed to halt, we know that $\overline{M}$ is also guaranteed to halt. Moreover, $\overline{M}$ accepts exactly those strings that M does not accept. Thus $\overline{M}$ accepts $\overline{L}$. ◀

By Theorem 9.2, we know that if a language L is RE, but $\overline{L}$ is not RE, then L cannot be recursive. For if L were recursive, then $\overline{L}$ would also be recursive and thus surely RE.

Theorem 9.3 *If both a language L and its complement are RE, then L is recursive. Not that then by Theorem 9.2, $\overline{L}$ is recursive as well.*

Proof Let $L = L(M_1)$ and $\overline{L} = L(M_2)$.

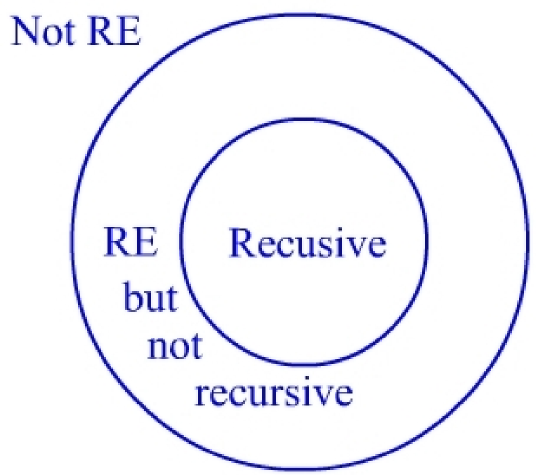


Both M_1 and M_2 are simulated in parallel by a TM M. We can make M a two-tape TM, and then convert it to a one-tape TM, to make the simulation easy and obvious.

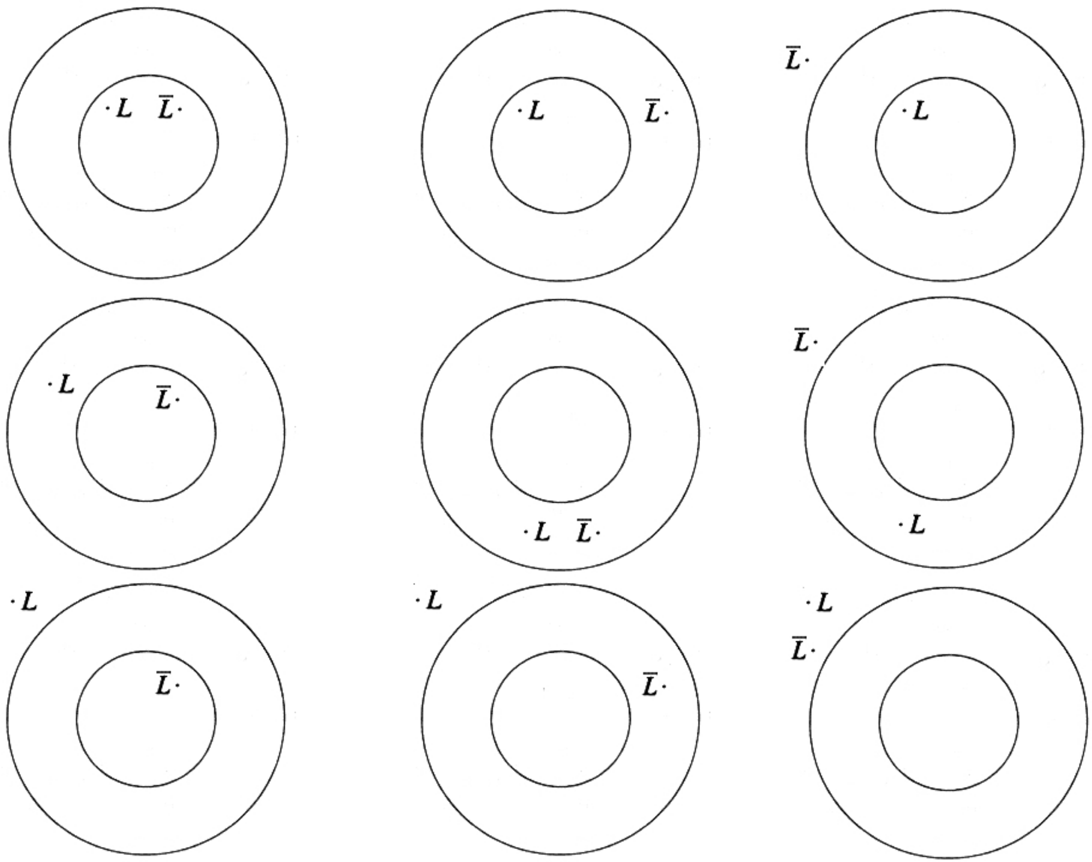
One tape of M simulates the tape of M_1, while the other tape of M simulates the tape of M_2. The states of M_1 and M_2 are each components of the state of M.

If input w to is in L, then M_1 will eventually accept. If so, M accepts and halts. If w is not in L, so M_2 will eventually accept. When M_2 accepts, M halts without accepting. Thus, on all inputs, M halts, and $L(M)$ is exactly L. So L is recursive. ◄

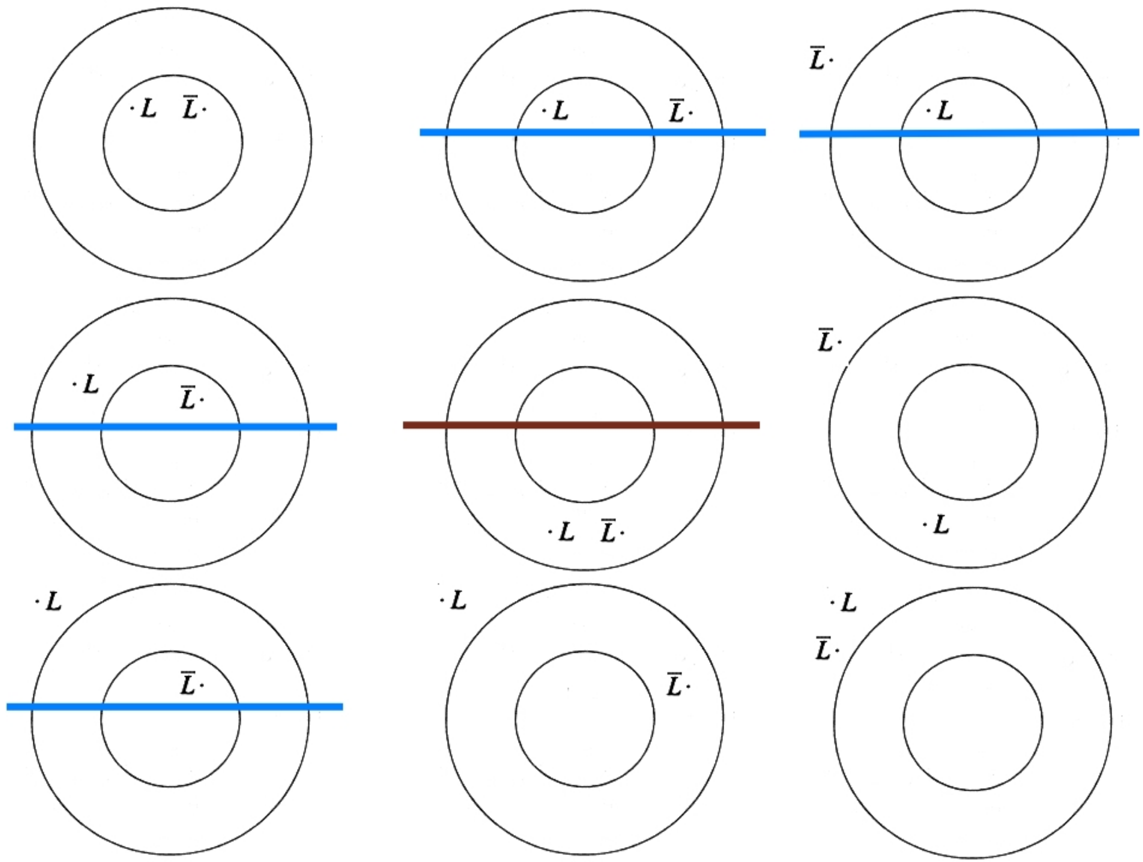
Summary: Of the nine possible ways to place a language L and its complement $\overline{L}$ in the diagram,



only the four are possible.

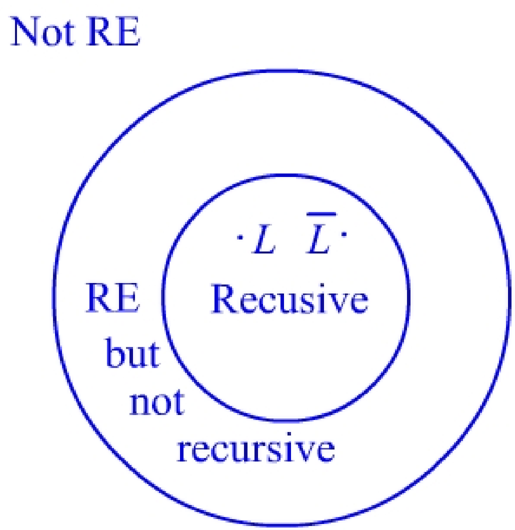


$\cdot L$ $\overline{L}\cdot$
$\cdot L$ $\overline{L}\cdot$
$\overline{L}\cdot$ $\cdot L$
$\cdot L$ $\overline{L}\cdot$
$\cdot L$ $\overline{L}\cdot$
$\overline{L}\cdot$ $\cdot L$
$\cdot L$ $\overline{L}\cdot$
$\cdot L$ $\overline{L}\cdot$
$\cdot L$ $\overline{L}\cdot$

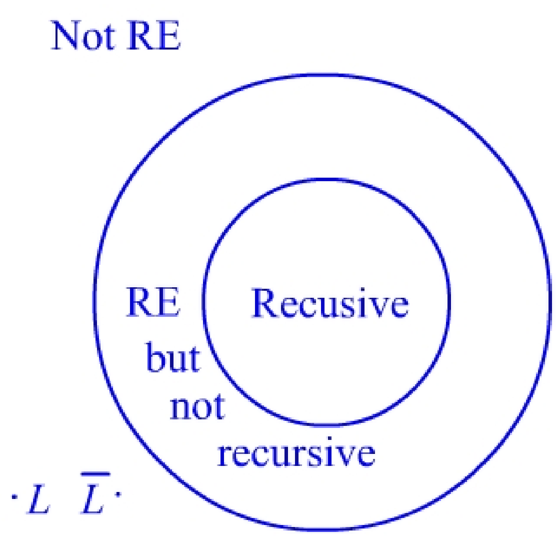


$\cdot L$ $\overline{L}\cdot$
$\cdot L$ $\overline{L}\cdot$
$\overline{L}\cdot$ $\cdot L$
$\cdot L$ $\overline{L}\cdot$
$\cdot L$ $\overline{L}\cdot$
$\overline{L}\cdot$ $\cdot L$
$\cdot L$ $\overline{L}\cdot$
$\cdot L$ $\overline{L}\cdot$
$\cdot L$ $\overline{L}\cdot$

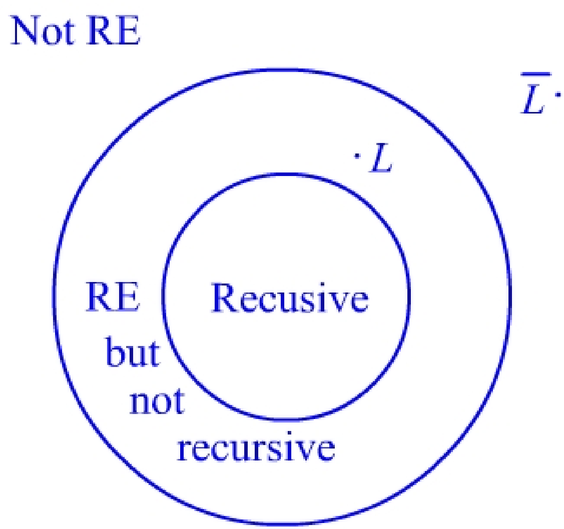
Case 1. Both L and $\overline{L}$ are recursive.



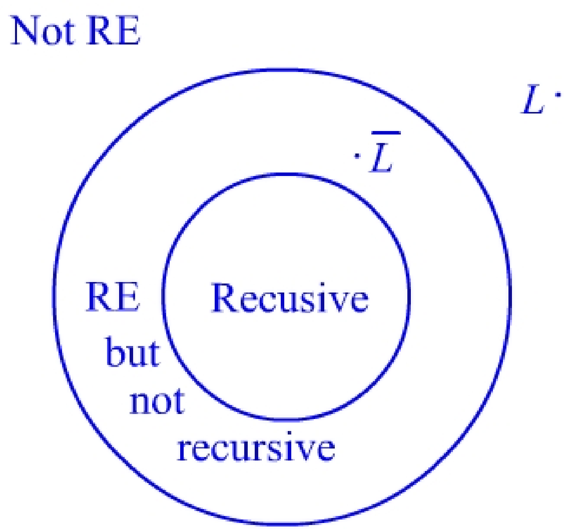
Case 2. Neither L nor $\overline{L}$ is RE.



Case 3. L is RE, but not recursive, and $\overline{L}$ is not RE.



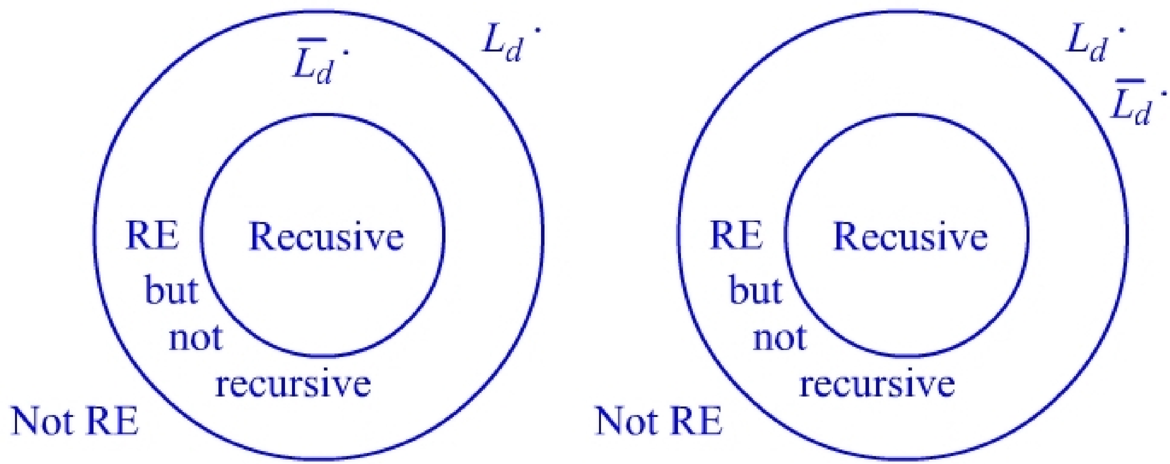
Case 4. $\overline{L}$ is RE, but not recursive, and L is not RE.



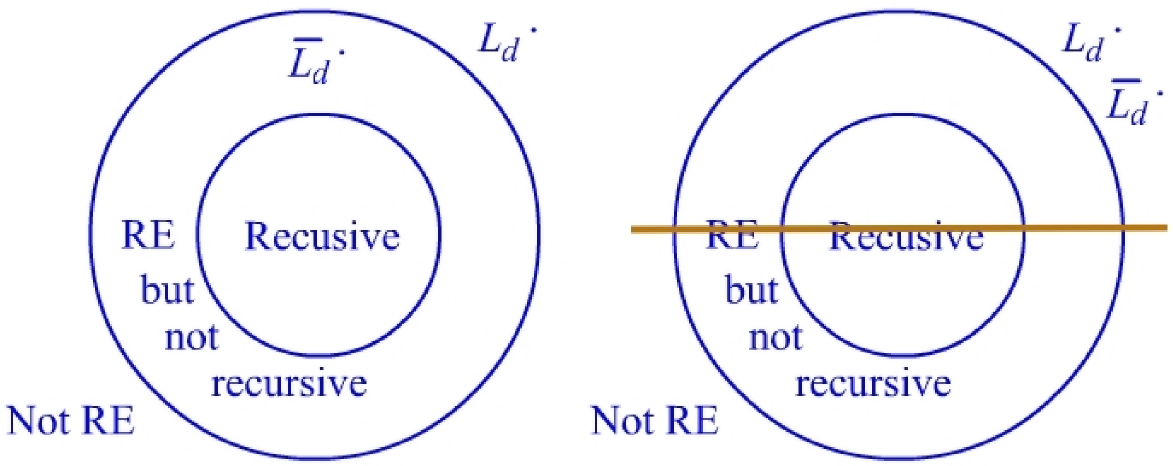
In proof of the above.

- Theorem 9.2 eliminates the possibility that one language (L or $\overline{L}$) is recursive and the other is in either of the other two classes.
- Theorem 9.3 eliminates the possibility that both are RE but not recursive.

Example As an example, consider L_d which is not RE. Thus, $\overline{L_d}$ could not recursive. It is, however, possible that $\overline{L_d}$ could be either non-RE or RE-but-not-recursive. It is in fact the latter.



$\overline{L_d}$.
L_d .
RE
but
not
recursive
Recusive
Not RE
L_d .
$\overline{L_d}$.
RE
but
not
recursive
Recusive
Not RE


$\overline{L_d}$
L_d
RE
but
not
recursive
Recusive
Not RE
L_d
$\overline{L_d}$
RE
but
not
recursive
Recusive
Not RE

The Universal Language

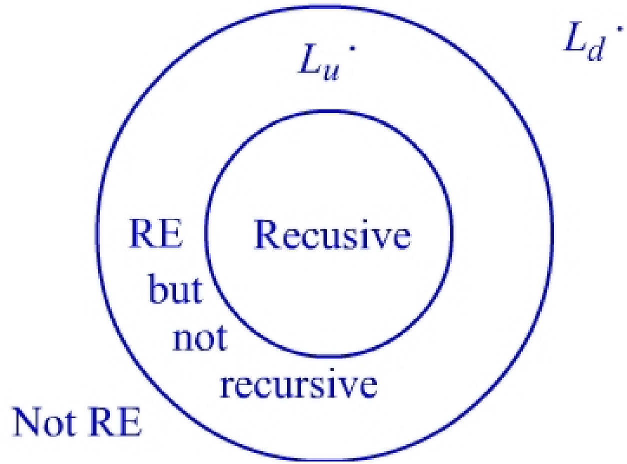
Now, we can make a vital definition

- The language L_u, the universal language, is the set of binary strings that encode a pair (M, w), where M is a TM with the binary input alphabet, and w is a string in $\{0, 1\}$, such that $w \in L(M)$

L_u is a language on alphabet $\{0, 1\}$, (M, w) is of the form $C_1 11 \cdots C_{n-1} 11 C_n 111 w$ where each of the C's is the code for one transition of M.

Theorem 9.4 *There is a Turing machine U, often called the universal Turing machine, such that $L_u = L(U)$. That is, L_u is a recursively enumerable language.*

We will give the proof later. Here, we first prove that L_u is not recursive! That is,

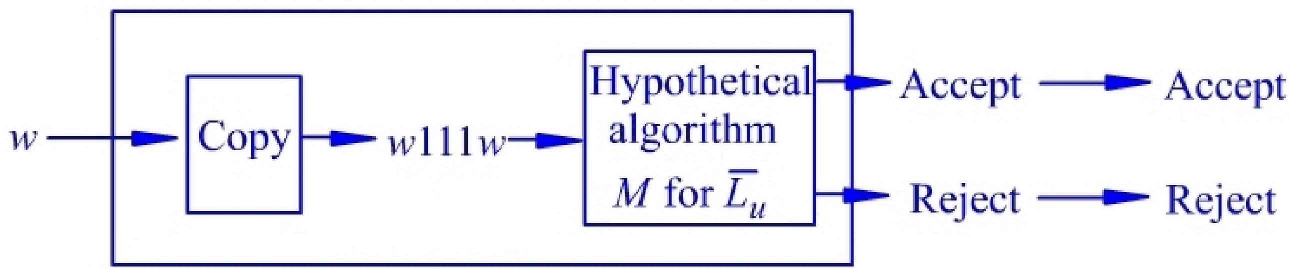


Theorem 9.5 *L_u is RE but not recursive. That is, if we think of the language L_u as a "problem", then problem L_u is undecidable.*

Proof Suppose L_u were recursive. Then by Theorem 9.2, $\overline{L_u}$ would also be recursive. However, if we have a TM M to accept $\overline{L_u}$, then we can construct a TM to accept L_d (by a method explained below).

Since we already know that L_d is not RE, we have a contradiction of our assumption that L_u is recursive.

Suppose $L(M) = \overline{L_u}$. As suggested by following figure



M' for L_d

we can modify TM M into a TM M' that accepts L_d as follows.

1. Given string w on its input, M' changes the input to $w111w$. We can write a TM program to do this step on a single tape. However, an easy argument that it can be done is to use a second tape to copy w, and then convert the two-tape TM to a one-tape TM.

2. M' simulates M on the new input. If w is w_i in our enumeration, then M' determines whether M_i accepts w_i. Since M accepts $\overline{L_u}$, it will accept if and only if M_i does not accept w_i; i.e., w_i is in L_d.

Thus, M' accepts w if and only if w is in L_d. Since we know M' cannot exist by Theorem 9.1, we conclude that L_u is not recursive. ◀

The proof of Theorem 9.4

Thank you!