Theory of Formal Languages (Recursive and Recursively Enumerable Languages)

Lecture 13: Feb. 16, 2019

Prof. K.R. Chowdhary *: Professor of CS*

13.1 Introduction

A language acceptable by Turing machine is called *Recursively Enumerable* (RE), which means that the set of strings in the language accepted by the Turing Machine can be enumerated. Recursively enumerable language is a type of formal language which is also called *partially decidable* or *Turing-recognizable.* It is known as *type-0* language in the Chomsky hierarchy of formal languages. The RE languages are always countably infinite. The class of RE languages has a broad coverage of languages, and they include some languages, which cannot be defined by a mechanical algorithm. TMs will fail to halt on some input not in these languages. If w is a string in RE language then the TM will eventually halt on w.

There exists three equivalent major definitions for the concept of a RE language.

Definition 13.1 RE Language. *A RE formal language is a recursively enumerable subset in the set of all possible words over the alphabet of the language.*

Definition 13.2 RE Language. *A RE language is a formal language for which there exists a Turing machine (or other computable function) which will enumerate all valid strings of the language.*

Note that, if the language is infinite, the enumerating algorithm provided can be chosen so that it avoids repetitions, since we can test whether the n-th string produced is "already" produced for some number less than n. If it is already produced, use the output for input $n + 1$ instead (recursively), but again, test whether it is "new".

Definition 13.3 RE Language. *A RE language is a formal language for which there exists a Turing machine (or other computable function) that will halt and accept when presented with any string in the language as input. But may either halt and reject or loop forever when presented with a string not in the language.*

Contrast this to **recursive** languages, which require that the Turing machine halts in all cases.

All regular, context-free, context-sensitive and recursive languages are recursively enumerable. The RE languages together with their complement *co-RE*, form the basis for the *arithmetical hierarchy.*

However, if M is still running on some input, we can never tell whether M will ultimately accept if it is allowed to run long enough or M will run forever. Therefore, it is appropriate to separate the subclass of RE languages accepted by at least one TM that halts on all inputs. However, halting may or may not be preceded by acceptance.

13.2 Recursive and Recursively Enumerable Languages

Definition 13.4 Recursive. *They allow a function to call itself. Or, a recursive language is a recursive subset in the set of all possible words over alphabet* Σ *of that language.*

A language is *Recursive (R)* if some Turing machine M recognizes it and halts on every input string, $w \in \Sigma^*$. Note that Recognizable is equal to Decidable. Or A language is recursive if there is a membership algorithm for it.

Let L be a *recursive* language and M the Turing Machine that accepts (i.e. recognizes) it. For string w, if $w \in L$, then M halts in final state. If $w \notin L$, then M halts in non-final state. (*halts* always!).

Non-recursive should not be taken as simpler version of computation, i.e., e.g., obtaining factorial value without recursion method. We have the relation:

$$\begin{aligned}\text{Regular languages} &\subseteq CF \text{ languages} \subseteq CS \text{ languages}\\ &\subseteq R \text{ languages} \subseteq RE \text{ languages}. \quad (13.1)\end{aligned}$$

Definition 13.5 Recursively Enumerable. *A language is* Recursively Enumerable *(RE) if some Turing machine accepts it.*

A Turing Machine M with alphabet Σ, *accepts* language L if,

$$L = \{w \mid w \in \Sigma^* \text{ and } M \text{ halts with input } w\}. \quad (13.2)$$

Let L be a RE language and M the Turing Machine that accepts it. Therefore, for $w \in L$, the TM M halts in final state, and for $w \notin L$, M halts in non-final state or *loops for ever.*

Relation between Recursive and RE Languages Every Recursive (R) language is Recursively Enumerable (RE). Therefore, if M is Turing Machine recognizing language L, which is R, then M can be easily modified so its accepts language that is RE. The languages which are *non-RE* cannot be recognized by TM. These are diagonal (L_d) languages of the diagonal of x, y, where x_i is language string w_i, and y_i is TM M_i. (see Fig. 13.1). Language $\langle M, w\rangle$, where M is TM and w is string, is not RE language, since its generalized form is not Turing decidable (undecidability proof), therefore, it is *non-RE* language. Every is recursive language can be enumerated. The following theorem proves this.

Theorem 13.6 *If a language L is recursive then there exists an enumeration procedure for it.*

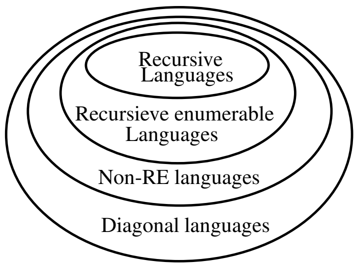

Figure 13.1: Relation between R and RE.

Proof: If $\Sigma = \{a, b\}$, then M' can enumerate strings: $a, b, aa, ab, ba, bb, aaa, \dots$.

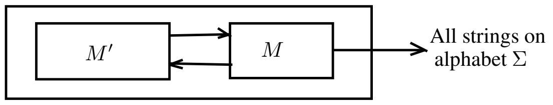

Figure 13.2: Enumeration of language.

The Enumeration procedure is as follows. Let M' generates string w. Next, the TM M checks, if $w \in L$ is yes, output w else ignore w. Let $L = \{a, ab, bb, aaa, \dots\}$. The M' output $= \{a, b, aa, ab, ba, bb, aaa, \}$, $L(M) = \{a, ab, bb, aaa, \dots\}$, and enumerated output is $a, ab, bb, aaa, \dots$ ■

13.3 TM as Language Enumerator

We can enumerate Turing machines, by encoding each one of them as a big integer in binary form (refer to chapter 12, Universal Turing machines). Assume that these big binary numbers which, each representing a Turing machine, are converted into decimal numbers. Then we may have large integer numbers representing, each, a Turing machine, which may be some what like, this:

- 50128478921234563456 is TM for balancing of parenthesis,
- 50256728933456723453 is TM for even number of 1's,
- 52563429393456734569 is TM for as Universal TM,
- 56239892129823489734 is TM which is equal Windows XP, and so on,

  ...

These numbers can also be represented in binary forms. Thus, a TM can be described by a set of 0s and 1s. This set forms a language, which may typically look, like,

$$\begin{aligned} L =\{&00010101010101010101010010101000111000, 00010101110110110101\\ &001001010100101, 00011101010110010111101011101001000, 000111\\ &010101010101010010101010010100, \dots\} \end{aligned}$$

This L is *countable set* of infinite number of strings, because, we have enumerated all those binary strings whose format is matching with the encoding of some TM. Their decimal equivalences are enumerated decimal number sequences. Thus, there is one-to-one correspondence between elements of the set of TMs and the corresponding natural numbers.

Let S be set of strings in decimal format. An enumeration procedure for S can be a TM E that generates all strings of S one-by-one, each in finite time, in lexicographic order. Let the s_i represent an element in S. Hence, $s_1, s_2, \cdots \in S$. Fig. 13.3 shows an enumeration machine, and we can have an algorithm to obtain the next sequence in the enumeration.

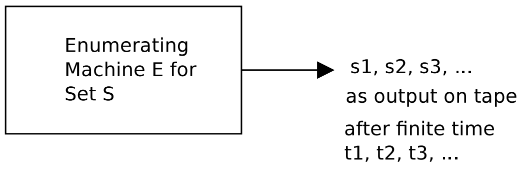

Figure 13.3: An Enumerator E for integer numbers as encodings of TMs.

Theorem 13.7 *If for a set there is a an enumeration procedure, then the set is countable. Prove that set of all the strings over $\Sigma = \{a, b, c\}$ is countable.*

Proof:

Repeat the following steps indefinitely:

1. Generate the next binary string of 1s and 0s in proper order,
2. Check if the string describes a Turing machine(an encoding of some TM),

i if yes: print the string on output tape,

ii if not, ignore it.

■

Theorem 13.8 *Set of All the Turing machines is countable.*

Proof: We know that any Turing machine can be encoded in binary strings of 0's and 1's. Next, the only requirement is to find an enumeration procedure for the set of TMs, which we already have as per theorem 13.7. ■

Example 13.9 *Turing Machine as language enumerator.*

The steps of enumerator as follows:

1. TM can also be designed to enumerate a language. Such machines produce the exhaustive list of of string of the language, progressively longer and longer.
2. Enumeration has no input, and its computation continues indefinitely if the language is infinite, as well as when it is finite.
3. For enumeration, a TM of tape $k \geq 2$ is used, tape-1 is output tape, which would hold all the generated strings, separated by #, and other tapes are working tapes.
4. Output tape-1 has: $B\#u_1\#u_2\# \dots \#u_i\# \dots$, where $u_i \in L$.
5. Tape head-1 always moves R, S, while others may move R, L, S.

Example 13.10 *Enumerate all the strings for* $L = \{a^n b^n c^n \mid n \geq 0\}$.

The L can be generated by two-tape TM E with following steps:

1. Write ## on tape-1 for ε.
2. add a on tape-2,
3. copy a equal to size of tape-2 on tape-1, then by same size the b is written on tape-1, then by same size c is written tape-1, followed with # written tape-1.
4. goto step-2.

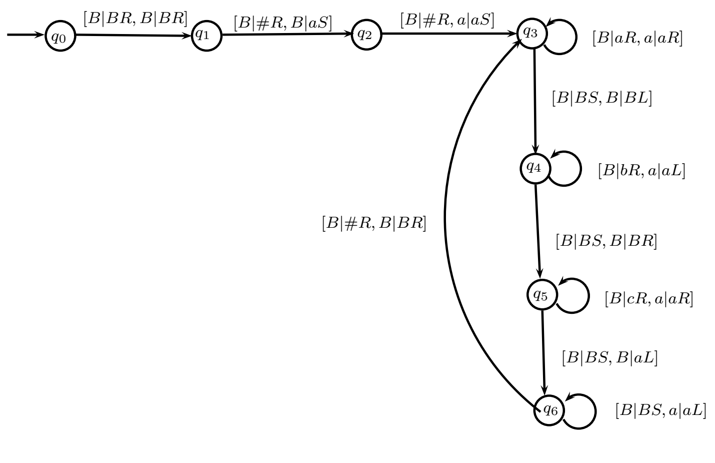

Figure 13.4: TM E as enumerator for the language $L = \{a^n b^n c^n \mid n \geq 0\}$.

The language $L = \{a^n b^n c^n \mid n \geq 0\}$ is suppose recognized by TM M. Let the enumerator E generates all the strings of L, shown in Fig. 13.4.

Theorem 13.11 *If L is enumerated by a TM then L is recursively enumerable.*

Proof: Let L is enumerated by a TM E of k tapes. We add a tape new $(k + 1)$th tape to it. Let this new machine is M. Consider a string $w \in L$, and we write it on $(k+1)$th tape. Every time symbol # is written on tape-1 by E, and it starts generating new string u_i, simultaneously it (i.e., u_i) is compared with w on tape $k + 1$, if found equal, M halts, otherwise the process of generating and compare is repeated for next string. Since, when $w \in L$ the machine M halts else continues indefinitely, L is recursively enumerable.

For enumeration it is necessary that lexicographic order (lo) of strings is generated. We can generate all the strings on alphabet $\Sigma = \{a_1, \ldots, a_n\}$ in lexicographic order using recursion as follows:

$$lo(\varepsilon) = 0,\ lo(a_i) = i, \text{ for } i = 1 : n$$

$$lo(a_i u) = i.n^{lenth(u)} + lo(u)$$

■

Theorem 13.12 *For any alphabet Σ, there is TM E_{Σ^*} that enumerates Σ^* in lexicographic order.*

Proof: Let M be the TM that accepts L. The lexicographic ordering produces listing: ε, u_1, u_2, $\ldots$, $\in \Sigma^*$. We construct a table with columns as strings in Σ^* and rows as natural numbers, in the order as shown in Fig. 13.5.

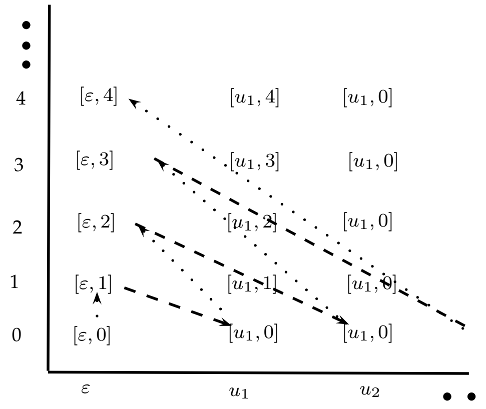

Figure 13.5: Enumeration in Lexicographic Order.

The $[i, j]$ entry in the table means "run the M for input u_i for j steps". The machine E is built to enumerate L such that enumeration of the ordered pairs are interleaved with the computation of M. The computation of E is a loop:

1. generate an ordered pair $[i, j]$
2. run a simulation of M with input u_i for j transitions or until the simulation halts.
3. If M accepts, write u_i on the output tape

4. continue with step 1.

If $u_i \in L$, the computation of M with input u_i halts and accepts after k transitions, for some number k. Thus, u_i will be written on output tape of E when ordered pair $[i, j]$ is processed. The 2nd element of k ensures that simulation is terminated after k steps. Consequently, no non-nonterminating computation are allowed, and each string of Σ^* is examined.

This is proof for one more theorem : "If a language is enumerated by TM then it is RE". ■

13.4 Countable and Uncountable Sets

Let a set of strings $S = \{s_1, s_2, \ldots, \}$ is countable. The s_i are generated through enumerating procedure. The Power set for S is 2^S, is not countable. Let some elements of power set are: $\{s_1\}, \{s_2, s_3\}, \{s_1, s_3, s_4\}$, etc. We can encode the elements of power set as binary strings of 1s and 0s. Let $t_1 = \{s_1\}$, $t_2 = \{s_2, s_3\}$, and so on (see table 13.1).

Theorem 13.13 *Power set is uncountable.*

Proof: Let us assume (to prove by contradiction) that power set is countable. Thus, we can enumerate its elements, as shown in table 13.1.

Table 13.1: Enumerating the power set elements.

Power set element	Power Set	Encoding				
		s_1	s_2	s_3	s_4	$\ldots$
t_1	$\{s_1\}$	1	0	0	0	$\ldots$
t_2	$\{s_1, s_3\}$	0	1	1	0	$\ldots$
t_3	$\{s_1, s_3, s_4\}$	1	0	1	1	$\ldots$
$\ldots$			$\ldots$		$\ldots$	

Take the power set element whose bits are the complement of diagonal, and let us call it x. This x is not t_1 because its first bit differs with t_1, it is also not t_2 because its second bit differs from t_2, and in general, it is not t_i because its ith element differs from t_i.

This new element must be some element t_i of power set (since we assume that $P(S)$ is enumerated). However, that is impossible. Hence, we conclude that power set is *uncountable*.

■

Countable TM v/s Uncountable Languages For $\Sigma = \{a, b\}$, the Σ^* is countable, because Σ^* can be enumerated, as $\Sigma^* = \{\epsilon, a, b, aa, ab, ba, bb, aaa, aab, \ldots\}$, which maps to $\{0, 1, 2, 3, \ldots\}$, hence countable, but infinite. However, the languages $\{L_1, L_2, \ldots\}$ that can be constructed from Σ^*, are subsets of 2^{Σ^*}, and are *uncountably infinite*.

On the other side, all the Turing machines $\{M_1, M_2, \ldots\}$ can be enumerated (refer to representation of all TMs), which is *countably infinite*. The conclusion is, there are more languages (which are uncountably infinite) than all the TMs (which are countably infinite), hence for some languages there does not exist TMs. In fact they are not *Turing recognizable.* What are those languages? We cannot say precisely, however, an example is, Russell's paradox.

13.5 Class of Languages

Following are the classes of languages:

- Recursive = decidable, their Turing Machines always halts.
- Recursive enumerable (semi-decidable) but not recursive = their Turing Machines always halt if they accept, otherwise halts in non-final state or loops.
- Non-recursively enumerable (*non-RE*) = there are no Turing Machine for them (see Fig. 13.1)

The Recursive languages are closed under complementation, the following theorem proves this.

Theorem 13.14 *If L is recursive then $\overline{L}$ is also recursive.*

Proof. Following are the steps:

1. The accepting states of M are made non-accepting states of M' with no transitions, i.e., here M' will halt without accepting.
2. If s is new accepting state in M', then there is no transition from this state.
3. If L is recursive, then $L = L(M)$ for some TM M that always halts. Transform M into M' so that M' accept when M does not and vice-versa. So M' always halts and accepts $\overline{L}$. Hence $\overline{L}$ is recursive.

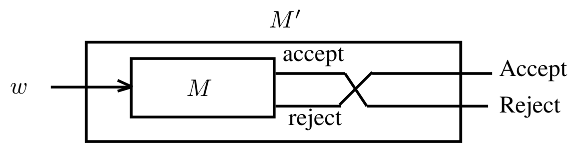

Figure 13.6: Co-Recursive is Recursive.

Theorem 13.15 *If L and $\overline{L}$ are* RE*, then L is Recursive.*

Proof: Let $L = L(M_1)$ and $\overline{L} = L(M_2)$. Construct a TM M that simulates M_1 and M_2 in parallel, using two tapes and two heads. If input to M is in L, then M_1 accepts it and halts, hence M accepts it and halts. Otherwise, if input to M is not in L, hence it is in $\overline{L}$, therefore, M_2 accepts and halts, hence M *halts* without accepting. Hence M halts on every input and $L(M) = L$. So L is recursive (see Fig. 13.7).

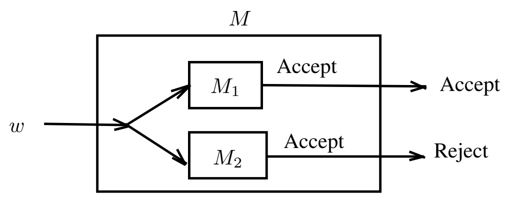

Figure 13.7: L and $\overline{L}$ are RE, then L is Recursive.

■

Closure Properties of Recursive Languages Set Recursive languages are closed under following set operations,

- Union,
- concatenation,
- intersection,
- Kleene star,
- Complement, and
- Set difference ($L_1 - L_2$).

Theorem 13.16 *A language L is recursive enumerable iff there exists an enumeration procedure for it.*

Proof: If there is an enumeration procedure, then we can enumerate all the strings, and compare each with w each time till it is found. If the language is RE, then we can follow an enumeration procedure to systematically generate all the strings (see Fig. 13.8).

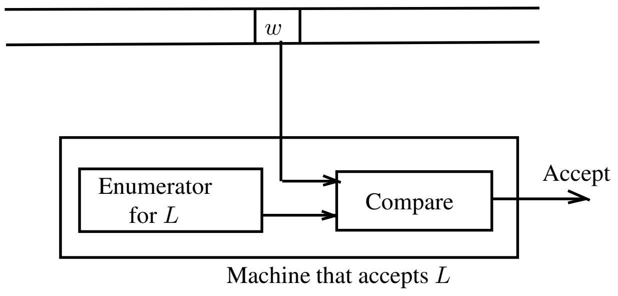

Figure 13.8: Enumeration machine for language L.

The algorithm for enumeration of RE language is as follows:

```
while (1) {
generate();
compare();
if same exit();
}
```

■

Intersection of *RE* and *R* Languages Given a *Recursive* and a *RE* languages: Their Union is *RE*, Intersection is *RE*, Concatenation is *RE*, and Kleenes closure is *RE*. If L_1 is *Recursive* and L_2 is *RE*, then $L_2 - L_1$ is *RE* and $L_1 - L_2$ is not *RE*.

Theorem 13.17 *The intersection R and* RE *languages is RE.*

Proof:

The proof is as follows.

1. Let L_1 and L_2 be languages recognized by Turing machines M_1 and M_2, respectively.
2. Let a new TM $M_\cap$ is for the intersection $L_1 \cap L_2$. $M_\cap$ simply executes M_1 and M_2 one after the other on the same input w: It first simulates M_1 on w. If M_1 halts by accepting it, $M\cap$ clears the tape, copies the input word w on the tape and starts simulating M_2. If M_2 also accepts w then $M_\cap$ accepts.
3. Clearly, $M_\cap$ recognizes $L_1 \cap L_2$, and if M_1 and M_2 halt on all inputs then also $M_\cap$ halts on all inputs.

■

Theorem 13.18 *The union of two Recursive languages is recursive.*

Proof:

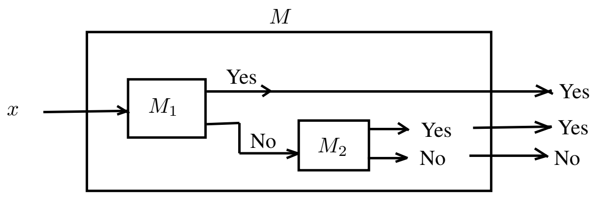

Figure 13.9: Union operation of Recursive Languages.

The TM corresponding to this must halt always. Let L_1 and L_2 be sets accepted by M_1 and M_2, respectively. Then $L_1 \cup Ł_2$ is accepted by TM M, where $x = w_1 \cup w_2$, for $w_1 \in L_1$ and $w_2 \in L_2$ (see Fig. 13.9). ■

Theorem 13.19 *The union of two* RE *languages is* RE.

Proof: Let L_1 and L_2 be sets accepted by M_1 and M_2, respectively. Then $L_1 \cup Ł_2$ is accepted by TM M, where $x = w_1 \cup w_2$, for $w_1 \in L_1$ and $w_2 \in L_2$.

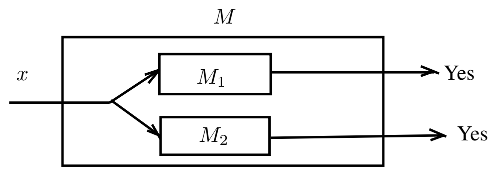

Figure 13.10: Union of two *RE* languages.

To determine if M_1 or M_2 accepts x we run both M_1 and M_2 simultaneously, using a two-tape TM M. The M simulates M_1 on the first tape and M_2 on the second tape. If either one enters the final state, the input is accepted (see Fig. 13.10). ■