A HIERARCHY OF FORMAL LANGUAGE AND AUTOMATA

Chap. 11

1

Summary

- The connection between *Turing Machines* and *Languages*.
- Depending on how one defines language acceptance, we get several different language families:
  - *Recursive Language (REC),*
  - *Recursively Enumerable Language (R.E.),* and
  - *Context-Sensitive Languages (CSL).*
- With *regular* and *context-free languages*, these languages form a properly nested relation called the *Chomsky Hierarchy*.

2

Learning Objectives

- Explain the *difference* between *Recursive (REC)* and *Recursively Enumerable Languages (R.E.).*
- Describe the type of *productions* in an *Unrestricted Grammar.*
- Identify the types of *languages* generated by *unrestricted grammars.*
- Describe the type of *productions* in a *Context Sensitive Grammar (CSG).*
- Give a sequence of derivations to generate a string using the productions in a *Context Sensitive Grammar*.
- Identify the types of *languages* generated by *Context-Sensitive Grammars.*
- Construct a Context-Sensitive Grammar to generate a particular language.
- Describe the structure and components of the *Chomsky Hierarchy.*

3

Recursive and Recursively Enumerable Languages

- Definition 11.1: A language L is *recursively enumerable* if there exists a Turing Machine that *accepts* it.

  (Note that the rejected strings cause the machine to either *not halt* or *halt in a nonfinal state*)

  i.e. there exists a TM M, s.t. $\forall w \in L$,

  $$q_0 w \vdash_M^* x_1 q_f x_2, \text{ with } q_f \in F.$$

- *Recursively Enumerable Language (R.E.)*

  = *Turing Acceptable Language.*

4

Recursive and Recursively Enumerable Languages

- Definition 11.2: A language L is *recursive* if there exists a TM that *accepts* it and is *guaranteed to halt* on *every valid input string*.

  i.e. for the *accepted* strings, TM halts in the final state

  while TM *halts in a non-final state* for the *rejected* strings.

  In other words, a language is *recursive* if and only if there exists a *membership algorithm* for it

  i.e. *decides* a membership.
- *Recursive Language = Turing Decidable Language*
- Claim:
  - Are there languages that are recursively enumerable but not recursive?
  - Are there languages, describable somehow, that are not recursively enumerable?

5

Cantor and Infinity (from Goddard's: chap. 14)

- Equal-Size Sets:
- If two *finite sets* are the equal size, one can pair the sets off:

  10 apples with 10 oranges.

  This is called a 1–1 correspondence: every apple and every orange is used up.
- So we say two *infinite sets* are the equal size if there exists a 1–1 correspondence.

6

Cantor and Infinity (from Goddard's: chap. 14)

- Countable Sets:
- Define N to be the set of all positive integers: {1, 2, 3, . . .}.
- The *even numbers* are the equal size as N:
  - one can pair 1 with 2, 2 with 4, 3 with 6, and so on.

  Note that the even numbers are used up: 1 – 2, 2 – 4, 3 – 6, ..
- A set is *countably infinite* if the equal size as N.
- It is *countable* if *finite* or *countably infinite*.
- This means there is a numbered list/enumeration of all elements.
- E.g.) The rational numbers are countable.
- But, there are sets that are *NOT countable*:

  uncountably infinite or uncountable: e.g.) The real numbers.

7

Cantor's Diagonalization (from Goddard's)

- Given a list of words of the same length, one can construct a word not on the list.
  - Start with the diagonal as a word, and then replace each letter by the next letter in the alphabet.
- Example:

1.	Q	U	I	E	T
2.	S	T	O	N	E
3.	O	F	F	E	R
4.	C	L	E	A	R
5.	P	H	L	O	X

  - The diagonal string is originally QTFAX.
  - Here diagonalization produces RUGBY. This is not on the list.

8

Cantor's Diagonalization (Chap.14@Goddard's)

- Example:

1.	Q	U	I	E	T
2.	S	T	O	N	E
3.	O	F	F	E	R
4.	C	L	E	A	R
5.	P	H	L	O	X

  - *Diagonalization*(QTFAX) = RUGBY $\notin$ the list.

- Diagonalization always gives new word.
  - The new word cannot be on the list: it is different from the 1st word in the 1st letter, different from the 2nd word in the 2nd letter, etc.
  - Cantor's insight was that same idea works with infinite lists.

9

Cantor's Theorem (Goddard's and Th^m 11.1@Lintz)

- Theorem 11.1: Let S be an countably infinite set.
  Then, its powerset 2^S (or $\wp(S)$) is not countable.

- Cantor's Theorem: The powerset $\wp(N)$ is *not countable*.

Proof by Contradiction)

Suppose $\wp(N)$ is countable. It means we can write down a list/enumeration of all the subsets of *S*.

Maybe the list starts: 1 – N, 2 – {4, 7}, 3 – {2, 4, 6, 8}, 4 – $\emptyset$, …

i.e. We have a function f: $N \rightarrow \wp(N)$ that maps numbers to subsets s.t. every subset appears in the list.

10

Cantor's Theorem (cont.)

- Cantor's Theorem: The powerset $\wp(N)$ is *not countable*.

Proof by Contradiction: cont.)

Now, define set T:

For each number $i \in N$, look up $f(i)$ and add i to T if $i \notin f(i)$.

But: T is not on list. T is not $f(1)$, because T and $f(1)$ differ on 1 (by definition $1 \in T \Leftrightarrow 1 \notin f(1)$).

And, T is not $f(2)$, because T and f(2) differ on 2, and so on.

That is, f is a lie; it does not use up the sets in $\wp(N)$.

This contradiction means: such a list does not exist.

i.e. There doesn't exist such a 1 – 1 function f.

Therefore, $\wp(N)$ is *not countable.* *Q.E.D.*

11

Cantor's Theorem (cont.)

Immediate Implication of Cantor's Theorem:

1) For any alphabet, the set of TMs is countable.
2) For any alphabet, the set of languages is uncountable.

- The set of TMs is countable because each TM can be represented by a binary number and hence as an integer.
- However, the subsets of the integers are not countable and hence the number of languages is uncountable.
- Therefore, there exists the languages that are *not* accepted by any TM, i.e. not recursively enumerable.

12

Languages that are Not Recursively Enumerable

- Theorem 11.2 : For any nonempty alphabet,
  there exist languages *not recursively enumerable*.
- Proof by *Diagonalization*, which can be used to show that there are fewer TMs than there are languages.
- More explicitly, Theorem 11.3 describes the existence of a recursively enumerable language whose *complement is not recursively enumerable*.
- Theorem 11.4: states a language and its complement is recursive; thus both are *r.e* as well as *rec.*

Furthermore,

- Theorem 11.5 concludes that the family of recursive languages is a proper subset of the family of recursively enumerable languages: $L_{REC} \subset L_{RE}$.

13

Languages that are Not Recursively Enumerable

- Theorem 11.2 : For any nonempty alphabet,
  there exist languages *not recursively enumerable*.

Proof) Any language L is a subset of Σ^*, and every such subset is a language. Thus, the size of a set of all languages is exactly $2^{|\Sigma^*|}$. Since Σ^* is infinite, the powerset of Σ^* ($\wp(\Sigma^*)$), i.e. the set of all languages on Σ, is not countable by Theorem 11.1 .

But, the set of all TMs can be enumerated and countable,
so the set of all recursively enumerable languages is countable.

Implication: There must be some languages on Σ that are
not recursively enumerable. Q.E.D.

14

Languages that are Not Recursively Enumerable

- Theorem 11.3: There exists a recursively enumerable language whose complement is not *RE*.: $L \in L_{RE}$, $\overline{L} \notin L_{RE}$

Proof) Let $\Sigma = \{a\}$. Consider the *set of all TMs* with Σ.
By Th^m 10.3, the set of all TM is countable, so we can order them, M_1, M_2, M_3, ... For each TM M_i , there is an associated recursively enumerable language $L(M_i)$. Conversely, for each recursively enumerable language on Σ, there is some TM that accepts it.
Now, consider a new language L defined as follows:
For each $i \geq 1$, the string $a^i \in L$ if and only if $a^i \in L(M_i)$.
It is clear that L is well defined, since the statement $a^i \in L(M_i)$, and hence $a^i \in L$, must be either true or false.
Next, consider the complement of L, $\overline{L}=\{a^i \mid a^i \notin L(M_i)\}$ (eq. 11.1) ,
which is well defined, but is not recursively enumerable.
Let's show it by contradiction.

15

Languages that are Not R.E.

- Theorem 11.3: There exists a recursively enumerable language whose complement is not recursively enumerable.

Proof. Cont.) Proof by Contradiction.
Let's assume that $\overline{L}$ is recursively enumerable,
Then, there exists a TM M_k , s.t. $\overline{L}$ = L(M_k). (eq. 11.2)
Consider the string a^k: $a^k \in L$ or $a^k \in \overline{L}$?
1) Suppose $a^k \in \overline{L}$. Then, it implies $a^k \in \overline{L}$ = L(M_k) (eq. 11.2) .
But, (eq. 11.1) implies $a^k \notin$ L(M_k) $=\overline{L}$ where ($\overline{L}=\{a^i \mid a^i \notin L(M_i)\}$ (eq. 11.1))
2) Suppose $a^k \in L$. Then, $a^k \notin \overline{L}$ and *(eq. 11.2)* implies $a^k \notin$ L(M_k)
But, (eq. 11.1) implies $a^k \in \overline{L}$. -- Contradiction!
So, the assumption $\overline{L}$ is r.e. is false.
Therefore, there exists a *R.E.* language whose complement is *not R.E.*

16

Languages that are Not R.E.

- Theorem 11.3: There exists a recursively enumerable language whose complement is not recursively enumerable.

Proof. Cont.) Further, Prove that *L* is *r.e*.

We can use the enumeration procedure for TMs.

Given a^i, we first find *i* by counting the number of *a*'s.

Then, use the enumeration procedure for TMs to find M_i.

Finally, we give its description along with a^i to a universal TM M_u that simulates the action of M on a^i.

If $a^i \in$ L, the computation carried out by M_u will eventually halt.

The combined effect of this is a TM that accepts every $a^i \in$ L.

Therefore, *L* is recursively enumerable

17

Language that is *R.E*. but *not Recursive*

- Theorem 11.4:

If a language *L* and $\overline{L}$ are both *r.e*., both *L* and $\overline{L}$ are *recursive*.
If *L* is recursive, then $\overline{L}$ is also recursive, and consequently both L and $\overline{L}$ are recursively enumerable.

Proof) If *L* and $\overline{L}$ are both *r.e*., there exists TMs M and M' that serves enumeration procedures for *L* and $\overline{L}$, respectively.

i.e. M enumerates $w_1, w_2, \ldots \in L$ and *M'* does $w_1', w_2', \ldots \in \overline{L}$.

Suppose $w \in \Sigma^+$. First let M generates w_1 and compare it with *w*.

If $w_1 \neq w$, let M' generates w_1', and compare it with *w*. Continue.

Any *w* will be generated by either *M* or *M'*, so eventually we match.

If the matching string is produced by M, $w \in$ L; otherwise, $w \in \overline{L}$.

This is a membership algorithm for both L and $\overline{L}$, so they are both recursive.

Next, assume *L* is recursive.

Then, there exists a membership algorithm for it. But, it also becomes a membership algorithm for $\overline{L}$ by complementing its conclusion.

Therefore, $\overline{L}$ is recursive.

Since any recursive language is recursively enumerable, Q.E.D.

18

Language that is *R.E.* but *not Recursive*

- Theorem 11.5: There exists a recursively enumerable language that is not recursive; that is,

  the family of *recursive* lang. $\subset$ the family of *r.e.* lang.

  Proof) Consider the language L of Th^m 11.3, s.t.

  L is r.e. but $\overline{L}$ is not.

  Therefore, by Th^m 11.4, L is not recursive.

  So, the family of recursive language is a proper subset of the family of *r.e.* language.

19

Unrestricted Grammars

- An *unrestricted grammar* has essentially *no restrictions* on the form of its productions:
  - *Any* variables and terminals on the *left/right* side, in any order.
  - The only restriction is that λ *is not allowed* on the *left* side of a production.
- Definition 11.3: A grammar $G = (V, T, S, P)$ is called *unrestricted* if all the productions are of the form

$$u \to v,$$

where $u \in (V \cup T)^+$ and $v \in (V \cup T)^*$.

- An Example of unrestricted grammar has productions:

$S \to S_1B$
$S_1 \to aS_1b$
$bB \to bbbB$
$aS_1b \to aa$
$B \to \lambda$

20

Unrestricted Grammars and Recursively Enumerable Languages

- Theorem 11.6: Any language generated by an *unrestricted grammar* is *recursively enumerable* (RE).
- Theorem 11.7: For every recursively enumerable language *L*, there exists an *unrestricted grammar G* that generates *L*.
- Conclusion:

  The *Unrestricted grammars generate exactly the family of Recursively Enumerable languages*,

  the largest family of languages that can be generated or recognized algorithmically.

21

Unrestricted Grammars & RE Languages (cont.)

- Theorem 11.6: Any language generated by an *unrestricted grammar* is *recursively enumerable* (RE).

Proof) The grammar defines a procedure for enumerating all strings in the language systematically. (Grammar ➔ TM)

For example, we can *list* all $w \in L$ such that $S \Rightarrow w$,

that is, *w* is derived in *one step*.

Since the set of the productions of the grammar is finite, there will be a finite number of such strings.

Next, we list all $w \in L$ that can be derived in *two steps,*

$S \Rightarrow x \Rightarrow w$, and so on.

We can simulate these derivations on a TM.

Thus, we have an enumeration procedure for the language. Hence, the generated language is recursively enumerable.

22

Unrestricted Grammars and RE Languages (cont.)

Proof) TM ➔Grammar: how TM can be mimicked by a grammar?

From a TM=(Q, Σ, Γ, δ,q_0, □, F), produce a grammar G s.t. L(G)=L(M).

Since the computation of the TM can be described by a sequence of ID

$$q_0 w \vdash^* x q_f y \text{ ,(eq.11.3)}$$

we will try to arrange it so that the corresponding grammar has the property that $q_0 w \Rightarrow^* x q_f y$ (eq.11.4) iff $q_0 w \vdash^* x q_f y$ holds.

Not so hard to do, but How to make connection

b/t $q_0 w \Rightarrow x q_f y$ and S $\Rightarrow^* w$ for all w satisfying $q_0 w \vdash^* x q_f y$,(eq.11.3) ?

To achieve this, we construct a grammar that has the following properties:

- step 1: S can derive $q_0 w$ for all $w \in \Sigma^+$, i.e. S $\Rightarrow^* w$
- step 2: $q_0 w \Rightarrow^* x q_f y$ (eq.11.4) is possible iff $q_0 w \vdash^* x q_f y$ (eq.11.3) holds.
- step 3: When a string $x q_f y$ with $q_f \in$F is generated, the grammar transforms this string $x q_f y$ into the original w.

Then, the complete sequence of derivation is:

$$S \Rightarrow^* q_0 w \Rightarrow^* x q_f y \Rightarrow^* w \quad \text{(eq.11.5)}$$

23

Unrestricted Grammars and RE Languages (cont.)

cont.) Issue: In step 3, how can the grammar remember w if it's modified during the step 2?

We can solve this by encoding strings so that the coded version originally has two copies of w. The first is saved, while the second is used in the steps in (eq.11.4). When a final configuration $x q_f y$ is entered, the grammar erases everything except the saved w.

To produce two copies of w and to handle the state symbol of M (which eventually has to be removed by the grammar), we introduce variables

V_{ab} and V_{aib} for all $a \in \Sigma \cup \{□\}$, $b \in \Gamma$, and all i such that $q_i \in Q$.

The variable V_{ab} encodes the two symbols a and b, while V_{aib} encodes a and b as well as the state q_i.

- 1st step: S $\Rightarrow^* q_0 w$ can be achieved (in the encoded form) by

$$S \rightarrow V_{□□}S \mid SV_{□□} \mid T, \quad (11.6)$$

$$T \rightarrow TV_{aa} \mid V_{a0a} \quad (11.7) \quad \text{for all } a \in \Sigma.$$

These productions allow the grammar to generate an encoded version of any string $q_0 w$ with an arbitrary number of leading and trailing blanks.

24

Unrestricted Grammars and RE Languages (cont.)

- <u>2^{nd} step</u>:
- For each transition $\delta(q_i, c) = (q_j, d, R)$ of M, put into productions:

  $V_{aic}V_{pq} \to V_{ad}V_{pjq}$ $\quad \forall a, p \in \Sigma \cup \{\square\}, \forall q \in \Gamma.$ (11.8)
- For each $\delta(q_i, c) = (q_j, d, L)$ of M, include it in G:

  $V_{pq}V_{aic} \to V_{pjq}V_{ad}$ $\quad \forall a, p \in \Sigma \cup \{\square\}, \forall q \in \Gamma.$ (11.9)
- If M enters a final state, the grammar must then get rid of everything except *w*, which is saved in the first indices of the V's.

  Thus, $\forall q_f \in$ F, we include productions: $V_{afb} \to a$ (11.10), $\forall a \in \Sigma \cup \{\square\}, \forall b \in \Gamma.$
- It creates the 1^{st} terminal in the string, which then causes a rewriting in the rest by: $cV_{ab} \to ca,$ (11.11) $V_{ab}\,c \to ac.$ (11.12)
- We need one more special production: $\square \to \lambda.$ (11.13)

  It takes care of the case when M moves outside that part of the tape occupied by the input *w*. To make it work, we must first use (11.6 & 7) to generate: $\square \ldots \square\ q_0 w\ \square \ldots \square$. The extra blanks are removed by (11.13).

25

Unrestricted Grammars and RE Languages (cont.)

- <u>Example 11.1</u>: Let M = $(Q, \Sigma, \Gamma, \delta, q_0, \square, F)$ be a TM where

  $Q = \{q_0, q_1\}, \Gamma=\{a, b, \square\}, \Sigma = \{a, b\}, F = \{q_1\}$ and

  $\delta(q_0, a) = (q_0, a, R), \delta(q_0, \square) = (q_1, \square, L).$

  Then, L(M) = L(*aa**).

  Consider the computation accepting $w=aa\square$:

  $q_0aa \vdash aq_0a \vdash aaq_0\square \vdash aq_1a\square$ (eq.11.14)

  To derive *aa* with G, use rules of the form (11.6) & (11.7)

  $$S \Rightarrow SV_{\square\square} \Rightarrow TV_{\square\square} \Rightarrow TV_{aa}V_{\square\square} \Rightarrow V_{a0a}V_{aa}V_{\square\square}.$$

  The last sentential form is the starting point for the part of the derivation that mimics the computation of the TM.

  It contains the original input $aa\square$ in the sequence of first indices and the initial ID, $q_0aa\square$ in the remaining indices.

  Next, we apply

26

Unrestricted Grammars and RE Languages (cont.)

- Example 11.1: Let M = (Q, Σ, Γ, δ, q_0, □, F) be a TM where
  Q = {q_0, q_1}, Γ={a, b, □}, Σ = {a, b}, F = {q_1} and
  $\delta(q_0, a) = (q_0, a, R)$, $\delta(q_0, \square) = (q_1, \square, L)$. Then, L(M) = L($aa$*).

cont.) Next, we apply $V_{a0a}V_{aa} \rightarrow V_{aa}V_{a0a}$, and $V_{a0a}V_{\square\square} \rightarrow V_{aa}V_{\square 0\square}$,
which are specific instances of (11.8), and

$V_{aa}V_{\square 0\square} \rightarrow V_{a1a}V_{\square\square}$ from (11.9).

Then, the next steps in the derivation are:

$$V_{a0a}V_{aa}V_{\square\square} \Rightarrow V_{aa}V_{a0a}V_{\square\square} \Rightarrow V_{aa}V_{aa}V_{\square 0\square} \Rightarrow V_{aa}V_{a1a}V_{\square\square}.$$

The sequence of first indices remains the same, always remembering the initial input. The sequence of the other indices is

$$0aa\square,\ a0a\square,\ aa0\square,\ a1a\square,$$

which is equivalent to the sequence of IDs in (11.14).

Finally, (11.10) to (11.13) are used in the last steps

$$V_{aa}V_{a1a}V_{\square\square} \Rightarrow V_{aa}aV_{\square\square} \Rightarrow V_{aa}a\square \Rightarrow aa\square \Rightarrow aa.$$

27

Unrestricted Grammars and RE Languages (cont.)

- Theorem 11.7: For every recursively enumerable language L, there exists an unrestricted grammar G, s.t. $L = L(G)$.

Proof) The construction described guarantees that

$$x \vdash y,$$

then $e(x) \Rightarrow e(y)$,

where $e(x)$ denotes the encoding of a string according to the given convention. By an induction on the number of steps, we can then show that $e(q_0w) \Rightarrow^* e(y)$

if and only if $q_0w \vdash^* y$.

We also must show that we can generate every possible starting configuration and that w is properly reconstructed if and only if M enters a final configuration. The details in Exercise 11.2-6.

28

Context-Sensitive Grammars (CSG)

- Definition 11.4: A *grammar* $G = (V, T, S, P)$ is said to be *context sensitive* if all productions are of the form

$$x \rightarrow y,$$

where $x, y \in (V \cup T)^+$ and $|x| \leq |y|$.

- i.e. only restriction in CSG is that, for any production, *length of the right side* is *at least as long as the length of the left side.*
- CSG is *noncontracting*, in the sense that in any derivation, the length of successive sentential forms can never decrease.
- These grammars are called *context-sensitive* because it is possible to specify that *variables may only be replaced in certain contexts*.

29

Context-Sensitive Languages (CSL)

- Definition 11.5: A *language L* is said to be *context sensitive* if there exists a *context-sensitive grammar G*, such that $L = L(G)$ or $L = L(G) \cup \{\lambda\}$.
- The empty string (λ) is included, because by definition, a CSG can never generate a language containing λ.
- As a result, it can be concluded that the family of Context-Free Language is a subset of the family of Context-Sensitive Languages: CFL $\subset$ CSL $\subset$ RE
- The language $\{ a^n b^n c^n \mid n \geq 1 \}$ is context-sensitive, since it is generated by the CSG in Ex. 11.2

30

CSL (cont.)

- Example 11.2: The CSG, G = (V, *T*, *S*, *P*) with productions

$$
\begin{aligned}
S &\rightarrow abc \mid aAbc \\
Ab &\rightarrow bA \\
Ac &\rightarrow Bbcc \\
bB &\rightarrow Bb \\
aB &\rightarrow aa \mid aaA.
\end{aligned}
$$

$L(G) = \{ a^n b^n c^n \mid n \geq 1 \}$.

A derivation of $a^3b^3c^3$:

$$
\begin{aligned}
S &\Rightarrow aAbc \Rightarrow abAc \Rightarrow abBbcc \Rightarrow aBbbcc \\
&\Rightarrow aaAbbcc \Rightarrow aabAbcc \Rightarrow aabbAcc \\
&\Rightarrow aabbBbccc \Rightarrow aabBbbccc \Rightarrow aaBbbbccc \\
&\Rightarrow aaabbbccc.
\end{aligned}
$$

- For instance, a variable *A* can only be replaced if it is followed by either *b* or *c*. -- context sensitive.

31

Derivation of Strings Using a CSG

- In Ex 11.2, the derivation in the CSG with productions

$$
\begin{aligned}
S &\rightarrow abc \mid aAbc \\
Ab &\rightarrow bA, && \text{(scan right)} \\
Ac &\rightarrow Bbcc \\
bB &\rightarrow Bb && \text{(scan left)} \\
aB &\rightarrow aa \mid aaA,
\end{aligned}
$$

$$S \Rightarrow aAbc \Rightarrow abAc \Rightarrow abBbcc \Rightarrow aBbbcc \Rightarrow aabbcc$$

- The variables A and B are effectively used as *messengers:*
  - an A is created on the left, travels to the *right* to the first c, where it creates another *b* and *c*, as well as variable B
  - the newly created *B* is sent to the *left* in order to create the corresponding *a*.

32

Context-Sensitive Languages (CSL) and Linear Bounded Automata (LBA)

- Theorem 11.8: For every CSL L *not including* λ, there is a *linear bounded automaton, M,* that recognizes L, i.e. $L = L(M)$.
- Theorem 11.9: If a language L is accepted by a *linear bounded automaton M*, then there is a CSG that generates L.
- Conclusion:

  CSGs generate exactly the family of languages accepted by LBA, the Context-Sensitive Languages.

33

Relationship between Recursive and Context-Sensitive Languages

- Theorem 11.10: Every Context-Sensitive language is recursive.
- Theorem 11.11: There exists a *Recursive Languages* that are not context-sensitive: CSL $\subset$ REC.

➔A *Hierarchical Relationship*

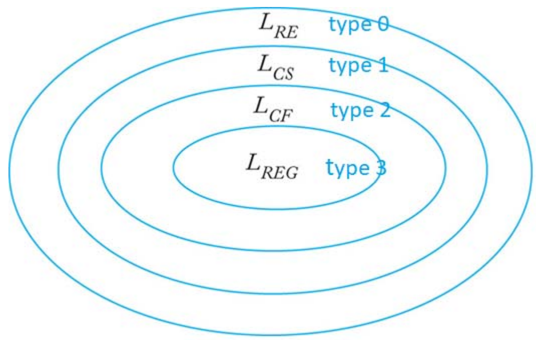
among the various classes of automata and languages:

- Linear bounded automata (LBA) are less powerful than Turing Machines (TM).
- Linear Bounded Automata are more powerful than Pushdown Automata.

34

The Chomsky Hierarchy

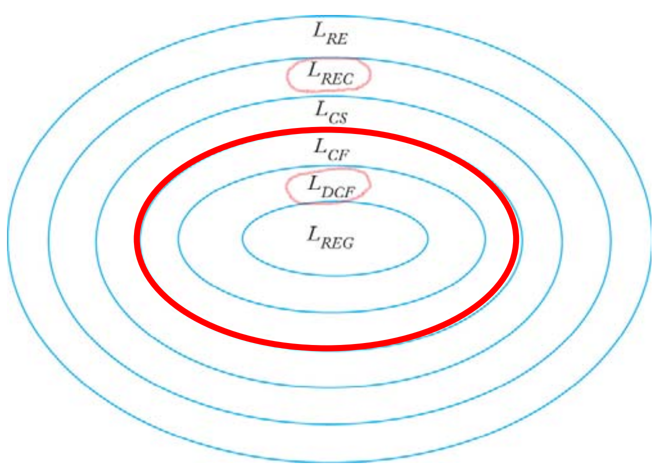
- The linguist *Noam Chomsky* summarized the relationship between language families by classifying them into four language types, type 0 to type 3.
- This classification is known as the *Chomsky Hierarchy*:



35

An Extended Hierarchy

- By including Deterministic Context-Free Languages and Recursive Languages, we get *the extended hierarchy.*



36

An Extended Hierarchy (cont.)

- Example 11.3: The CFL, $L = \{w \mid n_a(w) = n_b(w)\}$, was shown that it is *deterministic*, but *not linear* (Ex.8.5(B)).
  On the other hand, the language

  $$L = \{ a^n b^n \} \cup \{ a^n b^{2n} \}$$

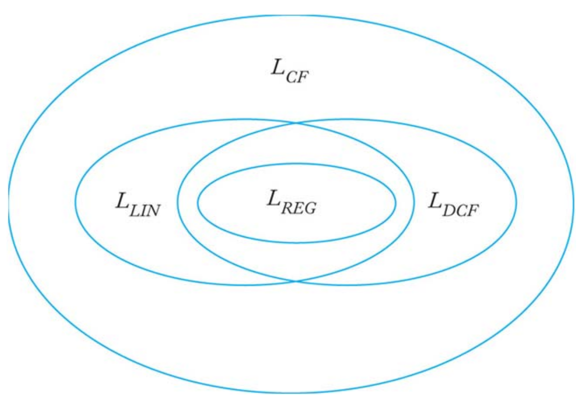
  is *linear*, but *not deterministic* (Ex.7.11).
  cf.) $\{ a^n b^n \}$ is a DCFL and linear (Ex.7.10 & 8.5(A))

➔ The Relationship between *regular, linear, deterministic context-free*, and *nondeterministic context-free languages* in Fig. 11.5.

37

A Closer Look at the Family of Context-Free Languages

The relationships among various subsets of the family of CFLs:
Regular (L_{REG}), *Linear* (L_{LIN}), *Deterministic Context-Free* (L_{DCF}), and *Nondeterministic Context-Free* (L_{CF})



38