COMPUTATIONAL THEORY

MODULE: REGULAR & CONTEXT-FREE GRAMMARS

TOPICS COVERED
✥ Grammars and Chomsky Hierarchy
✥ Regular Grammar (RG)
✥ Equivalence of Left and Right Linear Grammars
✥ Equivalence of RG and FA
✥ Context Free Grammars (CFG) – Definition & Derivations
✥ Parse Trees & Ambiguity
✥ Simplification of CFG
✥ Normal Forms: CNF and GNF
✥ Context Free Language (CFL) – Applications
✥ Pumping Lemma for CFL
✥ Closure Properties of CFL

University	Pattern	Subject
Mumbai University	NEP 2020	Computational Theory

1. Grammars and Chomsky Hierarchy

1.1 What is a Grammar?

A **grammar** is a formal mathematical system used to generate all strings of a language. It specifies the rules (productions) for forming valid strings. Formally, a grammar is defined as a **4-tuple G = (V, T, P, S)** where:

- **V** = Finite set of **Variables** (Non-terminals) — symbols that can be replaced
- **T** = Finite set of **Terminals** — actual symbols of the language ($V \cap T = \varnothing$)
- **P** = Finite set of **Production Rules** — rules of the form $\alpha \to \beta$
- **S** = **Start Symbol** where $S \in V$ — derivation begins here

1.2 Chomsky Hierarchy

Noam Chomsky (1956) classified all formal grammars into four hierarchical types based on the form of their production rules. Each type is strictly more powerful than the next.

Type	Grammar Name	Language Class	Automaton Recognizer	Production Form
Type 0	Unrestricted	Recursively Enumerable	Turing Machine	$\alpha \to \beta$ (no restriction)
Type 1	Context-Sensitive (CSG)	Context-Sensitive	Linear Bounded Automaton	$\alpha A\beta \to \alpha\gamma\beta$, $\|\alpha\| \le \|\beta\|$
Type 2	Context-Free (CFG)	Context-Free	Pushdown Automaton (PDA)	$A \to \alpha$ (A single non-terminal)
Type 3	Regular Grammar (RG)	Regular	Finite Automaton (FA)	$A \to aB$ or $A \to a$

Key Relationship

The hierarchy is strictly nested: **Type 3** ⊂ **Type 2** ⊂ **Type 1** ⊂ **Type 0**. Every Regular Language is a CFL, every CFL is Context-Sensitive, and so on. The converse is not true — there exist CFLs that are not regular.

2. Regular Grammar (RG)

2.1 Definition

A **Regular Grammar** (Type 3) generates exactly the class of Regular Languages. It exists in two symmetric forms:

Right Linear Grammar (RLG)

Every production has one of the forms:

- **A → aB** (terminal followed by non-terminal)
- **A → a** (terminal only)
- **A → ε** (empty string, for nullable variables)

Left Linear Grammar (LLG)

Every production has one of the forms:

- **A → Ba** (non-terminal followed by terminal)
- **A → a** (terminal only)
- **A → ε** (empty string)

2.2 Example of Right Linear Grammar

Grammar G for L = { $a^n b$ | n ≥ 0 }:

```
S → aS | b
```

Derivation of string "aab":

```
S ⇒ aS ⇒ aaS ⇒ aab ✓
```

2.3 Equivalence of Left and Right Linear Grammars

Theorem: Left Linear Grammars and Right Linear Grammars are equivalent in expressive power. Both generate exactly the class of Regular Languages.

Proof Sketch (RLG → LLG):

- Step 1: Given RLG G generating L, construct a grammar G' by reversing all productions (e.g., A → aB becomes B → Aa).
- Step 2: The reversed grammar G' generates L^R (the reversal of L) and is a Left Linear Grammar.
- Step 3: Regular languages are closed under reversal, so L^R is also regular.
- Step 4: By symmetry, any LLG can similarly be converted to an RLG.
- Conclusion: LLG and RLG generate the same language class. ■

Example Conversion:

```
Right Linear: S → aA, A → bS, A → b
Reversed LLG: A → Sa, S → Ab, S → b
```

```
Both generate: { (ab)^n · b | n ≥ 0 }
```

3. Equivalence of Regular Grammar and Finite Automata

Theorem: A language L is Regular if and only if there exists a Regular Grammar G such that L = L(G). This establishes the equivalence: **RG** ↔ **NFA** ↔ **DFA**.

3.1 Converting Right Linear Grammar → NFA

Given RLG G = (V, T, P, S), construct NFA M = (Q, Σ, δ, q■, F):

- **States Q:** One state per non-terminal in V, plus one additional final state **f**
- **Start state:** q■ = S
- **Final states:** F = { f }
- **For A → aB:** add transition δ(A, a) = B
- **For A → a:** add transition δ(A, a) = f
- **If S → ε:** add S to F (ε ∈ L)

Example:

```
Grammar: S → aS | bA, A → b

NFA States: { S, A, f }
δ(S, a) = S (from S → aS)
δ(S, b) = A (from S → bA)
δ(A, b) = f (from A → b)
Start: S, Final: { f }
Language: { a^n · bb | n ≥ 0 }
```

3.2 Converting NFA → Right Linear Grammar

Given NFA M = (Q, Σ, δ, q■, F), construct RLG G = (V, T, P, S):

- **Variables V:** One non-terminal per state in Q
- **Start symbol S:** corresponds to start state q■
- **For each δ(q■, a) = q■:** add production q■ → a q■
- **For each δ(q■, a) = q■ where q■ ∈ F:** also add q■ → a
- **If q■ ∈ F:** add S → ε

4. Context-Free Grammar (CFG)

4.1 Definition

A **Context-Free Grammar (CFG)** is a 4-tuple **G = (V, T, P, S)** where every production has the form **A** → α, with A a single non-terminal and α ∈ (V ∪ T)*. Called 'context-free' because non-terminal A can be replaced by α in ANY context.

Example: Grammar for L = { $a^n b^n$ | n ≥ 1 }:

```
S → aSb | ab
```

Derivation of "aaabbb":

```
S ⇒ aSb ⇒ aaSbb ⇒ aaabbb ✓
```

4.2 Sentential Forms

Any string α ∈ (V ∪ T)* derivable from S (i.e., S ⇒* α) is a **sentential form**. It may contain both terminals and non-terminals. A sentential form that contains only terminals (α ∈ T*) is called a **sentence** of the language L(G).

4.3 Leftmost and Rightmost Derivations

Leftmost Derivation (LMD)

At every derivation step, the **leftmost non-terminal** in the sentential form is replaced. This corresponds to how top-down (LL) parsers work.

Rightmost Derivation (RMD)

At every derivation step, the **rightmost non-terminal** in the sentential form is replaced. This corresponds to how bottom-up (LR) parsers work.

Example: Grammar: E → E+T | T, T → T*F | F, F → id

For string "id + id * id":

```
LMD: E ⇒ E+T ⇒ T+T ⇒ F+T ⇒ id+T ⇒ id+T*F ⇒ id+F*F ⇒ id+id*F ⇒ id+id*id
RMD: E ⇒ E+T ⇒ E+T*F ⇒ E+T*id ⇒ E+F*id ⇒ E+id*id ⇒ T+id*id ⇒ F+id*id ⇒ id+id*id
```

5. Parse Tree (Derivation Tree)

A **parse tree** is a hierarchical, tree-based graphical representation of a derivation. It captures the structure of how a string is generated by a CFG.

5.1 Properties of a Parse Tree

- The **root** is labeled with the start symbol S
- Each **internal node** is labeled with a non-terminal
- Each **leaf** is labeled with a terminal or ε
- If an internal node has label A and children X■, X■, …, X■ (left to right), then A → X■X■…X■ is a production in P
- The **yield** (reading leaves left to right) gives the derived string
- One parse tree corresponds to exactly ONE leftmost derivation and ONE rightmost derivation

5.2 Example Parse Tree

Grammar: E → E+E | E*E | id. Parse tree for "id + id * id" (one interpretation):

```
E
/ | \
E + E
| / | \
id E * E
| |
id id

This represents: id + (id * id)
```

6. Ambiguity in CFG

6.1 Definition of Ambiguous Grammar

A CFG G is **ambiguous** if there exists at least one string w ∈ L(G) for which there are **two or more distinct parse trees** (equivalently, two or more distinct leftmost derivations, or two or more distinct rightmost derivations).

Ambiguity is a Property of the Grammar, Not the Language

A language may be generated by both ambiguous and unambiguous grammars. We call a CFL **inherently ambiguous** only if every possible CFG for it is ambiguous — i.e., no unambiguous grammar exists.

6.2 Removing Ambiguity — Arithmetic Expressions

The grammar E → E+E | E*E | id is ambiguous. Ambiguity is removed by encoding operator **precedence** (multiplication before addition) and **left-associativity**:

```
E → E + T | T (E handles addition, lowest precedence)
```

```
T → T * F | F (T handles multiplication)
F → id | ( E ) (F handles atoms and grouped expressions)
```

This restructured grammar is **unambiguous**. Each string has exactly one parse tree, correctly reflecting the intended evaluation order.

7. Simplification of CFG

Simplification removes redundancy and prepares a CFG for conversion to normal forms. The standard order is: (1) Remove ε-productions, (2) Remove unit productions, (3) Remove useless symbols.

7.1 Eliminating ε-Productions (Null Productions)

An **ε-production** is any production A → ε. A variable A is **nullable** if A ⇒* ε.

Algorithm:

- Step 1: Find all nullable variables (those that can derive ε).
- Step 2: For each production with nullable variables, add all combinations with and without each nullable variable on the RHS.
- Step 3: Delete all ε-productions. Retain S → ε only if ε ∈ L(G).

Example:

```
Original: S → AB, A → aA | ε, B → bB | ε

Nullable: A, B (both can derive ε)

New S → AB: S → AB | A | B (omit S→ε if ε ∉ L)
New A → aA: A → aA | a
New B → bB: B → bB | b

Result (if ε ∉ L):
S → AB | A | B
A → aA | a
B → bB | b
```

7.2 Eliminating Unit Productions

A **unit production** is of the form **A → B** where B ∈ V (single non-terminal on RHS). Unit productions create chains that can be collapsed.

Algorithm:

- Step 1: Find all unit pairs (A, B) such that A ⇒* B using only unit productions.
- Step 2: For each unit pair (A, B) and each non-unit production B → α, add production A → α.
- Step 3: Remove all unit productions from P.

Example:

```
Original: S → A, A → B, B → a | b

Unit pairs: (S,A), (S,B), (A,B)

Add: S → a | b (from B→a|b via S⇒*B)
A → a | b (from B→a|b via A⇒*B)

Remove: S→A, A→B

Result: S → a | b, A → a | b, B → a | b
```

7.3 Eliminating Useless Symbols and Productions

A symbol X is **useful** if it is both **generating** and **reachable**:

- **Generating:** X $\Rightarrow^*$ w for some terminal string w $\in$ T*
- **Reachable:** S $\Rightarrow^*$ $\alpha X \beta$ for some $\alpha, \beta \in (V \cup T)^*$

Algorithm (order matters!):

- Step 1: Find all generating symbols. Remove non-generating symbols and their productions.
- Step 2: From the remaining grammar, find all reachable symbols. Remove non-reachable symbols.

Example:

```
Original: S → AB | a, A → b, B → CC, C → DD

Step 1 (Generating):
D → ? (no production) → D is non-generating
C → DD → C is non-generating
B → CC → B is non-generating
A → b ✓, S → a ✓
Remove B, C, D and their productions

After Step 1: S → a, A → b

Step 2 (Reachable from S): S → a (A is not reachable now)
Remove A

Final Result: S → a
```

8. Normal Forms

8.1 Chomsky Normal Form (CNF)

A CFG is in **Chomsky Normal Form** if every production is of the form:

CNF Production Rules

- **A → BC** (exactly two non-terminals)
- **A → a** (exactly one terminal)
- **S → ε** (only if ε ∈ L(G))

Significance: CNF is used in the CYK (Cocke-Younger-Kasami) parsing algorithm, and is the standard form for many theoretical proofs about CFLs.

Step-by-Step Conversion to CNF

Step	Action	Example
1	Eliminate ε-productions	A → ε removed; new productions added
2	Eliminate unit productions	A → B replaced by A → α for B → α
3	Eliminate useless symbols	Non-generating/non-reachable removed
4	Replace terminals in long rules	S → aBC becomes S → T■BC, T■ → a
5	Break productions with 3+ symbols	A → B■B■B■ becomes A → B■X, X → B■B■

Full Example:

```
Original: S → aBC | b, B → AC, C → c

Step 4 — Replace terminals in long RHS:
S → T■BC | b where T■ → a
B → AC (already OK)
C → c (already OK)

Step 5 — Break 3-symbol RHS:
S → T■X where X → BC

CNF Result:
S → T■X | b
X → BC
B → AC
C → c
T■ → a
```

8.2 Greibach Normal Form (GNF)

A CFG is in **Greibach Normal Form** if every production is of the form:

GNF Production Rule

- **A → aα** where a ∈ T (a terminal) and α ∈ V* (zero or more non-terminals)
- Every RHS starts with exactly one terminal, followed by any number of non-terminals.

Significance: GNF guarantees that each derivation step consumes exactly one terminal. This means a string of length n requires exactly n steps, eliminating left recursion naturally. GNF is the basis for constructing PDAs from CFGs directly.

Key Steps for GNF Conversion

- Step 1: Convert to CNF (or at least eliminate ε, unit, useless productions)
- Step 2: Order non-terminals as A■, A■, …, A■
- Step 3: Ensure A■ → A■α only if j > i (by substitution — eliminate lower-index left non-terminals)
- Step 4: Eliminate any remaining left recursion: for A → Aα | β, replace with A → βA', A' → αA' | α
- Step 5: Back-substitute so every production begins with a terminal

Example:

```
Original: S → AA | b, A → SS | a

After ordering and substitution into GNF:
S → aAAS' | bAS' | aA | b (each starts with terminal)
A → aAAS' | bAS' | aA | b | ...
S' → AA S' | AA (new variable for left recursion removal)
```

9. Context-Free Language (CFL) — Applications

9.1 Parsers in Compilers

CFGs are the fundamental tool for defining the **syntax of programming languages**. Every language specification (C, Java, Python) includes a CFG. Compilers use parsers built from these CFGs to analyze source code.

Parser Type	Derivation Used	Grammar Class	Example Tools
Top-Down (LL)	Leftmost Derivation	LL(1) grammars	Recursive Descent, ANTLR
Bottom-Up (LR)	Rightmost Derivation (reverse)	LR(0), SLR, LALR, CLR	yacc, bison, GCC

The **parse tree** built during parsing serves as the **Abstract Syntax Tree (AST)**, which drives semantic analysis, type checking, and code generation.

9.2 Markup Languages

HTML and **XML** have hierarchical, nested tag structures that are inherently context-free. Tags must be properly nested — every opening tag must have a matching closing tag in the correct order.

```
Example: [b] [i] text [/i] [/b] <- Valid (properly nested)
[b] [i] text [/b] [/i] <- Invalid (improperly nested)
```

The nesting constraint (matching pairs) cannot be handled by regular grammars. XML parsers use CFG-based parsers (SAX, DOM) to validate document structure.

10. Pumping Lemma for Context-Free Languages

The Pumping Lemma for CFLs is used to **prove that a language is NOT context-free**. It is a necessary condition: every CFL satisfies it, so if a language violates it, it cannot be a CFL.

10.1 Statement of the Pumping Lemma

Pumping Lemma for CFL

If L is a Context-Free Language, then there exists a constant **p** (pumping length) such that for every string **$w \in L$** with **$|w| \geq p$**, we can write **w = uvxyz** satisfying all three conditions:

- **Condition 1:** $|vxy| \leq p$ (the middle section is not too long)
- **Condition 2:** $|vy| \geq 1$ (v and y cannot both be empty)
- **Condition 3:** $uv^nxy^nz \in L$ for all $n \geq 0$ (pumping preserves membership)

10.2 Strategy for Proving a Language is NOT CFL

Proof by Contradiction:

- Assume L is a CFL with pumping length p
- Choose a specific string $w \in L$ with $|w| \geq p$ (choose w carefully to force a contradiction)
- Show that for ALL ways to split w = uvxyz satisfying Conditions 1 and 2, there exists some n such that $uv^nxy^nz \notin L$
- This contradicts the Pumping Lemma, so L is NOT a CFL

10.3 Worked Example

Prove: $L = \{ a^nb^nc^n \mid n \geq 1 \}$ is not a CFL

```
Assume L is CFL with pumping length p.

Choose: w = a^p b^p c^p ( |w| = 3p ≥ p ✓ )

For ANY split w = uvxyz with |vxy| ≤ p and |vy| ≥ 1:
Since |vxy| ≤ p, the substring vxy cannot span all three blocks.
So v and y together contain at most two distinct symbols.

Case 1: v and y contain only a's and b's.
Pumping (n=2): count of c's stays the same, but a's or b's increase.
→ uv²xy²z has unequal counts → ∉ L. Contradiction!

Case 2: v and y contain only b's and c's. (Similar argument)
→ uv²xy²z: a's unchanged but b's/c's increase → ∉ L. Contradiction!

All cases lead to contradiction.
∴ L = { a^n b^n c^n } is NOT a CFL. ■
```

Exam Tip: Choosing w

Always choose w to be the 'most balanced' or 'most symmetric' string, like $a^pb^pc^p$. This forces the split vxy to be confined to at most two symbol types, making the contradiction easy to derive.

11. Closure Properties of Context-Free Languages

Closure properties tell us whether performing an operation on CFL(s) always produces another CFL. These properties are essential for proving languages are or are not context-free.

Operation	CFLs Closed?	Proof Method / Notes
Union L■ ∪ L■	■ YES	New start S → S■ | S■; combine grammars
Concatenation L■ · L■	■ YES	New start S → S■S■; combine grammars
Kleene Star L*	■ YES	New start S → SS■ | ε; combine grammars
Reversal L^R	■ YES	Reverse every production's RHS
Homomorphism h(L)	■ YES	Replace terminals via PDA construction
Intersection L■ ∩ L■	■ NO	Counterexample shown below
Complement L■	■ NO	Would imply closure under intersection
Difference L■ - L■	■ NO	L■ - L■ = L■ ∩ L■■
Intersection with Regular L ∩ R	■ YES	Product construction: PDA × DFA

11.1 Key Proofs

Union (Closed)

Given G■ = (V■, T, P■, S■) and G■ = (V■, T, P■, S■) with V■ ∩ V■ = ∅, construct G = (V■ ∪ V■ ∪ {S}, T, P■ ∪ P■ ∪ {S → S■ | S■}, S). Then L(G) = L■ ∪ L■, and G is a valid CFG. ■

Intersection NOT Closed — Counterexample

```
L■ = { a^n b^n c^m | n,m ≥ 0 } is a CFL (grammar: S → AB, A → aAb | ε, B → cB | ε)
L■ = { a^m b^n c^n | n,m ≥ 0 } is a CFL (grammar: S → AB, A → aA | ε, B → bBc | ε)

L■ ∩ L■ = { a^n b^n c^n | n ≥ 0 } which is NOT a CFL (shown by pumping lemma!)

∴ The intersection of two CFLs need not be a CFL. ■
```

Intersection with Regular Language (Closed)

If L is a CFL accepted by PDA P, and R is a regular language accepted by DFA D, construct a product automaton (PDA × DFA) that simulates both simultaneously. This product machine is also a PDA, and it accepts L ∩ R. Therefore L ∩ R is a CFL. ■

Important Application

Intersection with regular languages is very useful in proofs. To show a CFL L is not regular, we can intersect it with a suitable regular language and apply the pumping lemma for regular languages to the result.

12. Quick Reference Summary

Concept	Key Definition / Property
Grammar G = (V,T,P,S)	V=variables, T=terminals, P=productions, S=start
Chomsky Type 3 (Regular)	A → aB or A → a; recognized by FA
Chomsky Type 2 (CFG)	A → α; recognized by PDA
LLG ≡ RLG	Both generate exactly the Regular Languages
RG ↔ FA	Directly and mutually convertible
Sentential Form	Any string derivable from S (may contain non-terminals)
LMD	Always replace leftmost non-terminal first
RMD	Always replace rightmost non-terminal first
Parse Tree Yield	Read leaves left-to-right to get derived string
Ambiguity	Exists if any string has ≥ 2 parse trees
Inherent Ambiguity	Every CFG for that language is ambiguous
ε-productions	Remove by finding all nullable variables
Unit productions	A→B; remove by finding unit pairs and short-circuiting
Useless symbols	Remove non-generating first, then non-reachable
CNF	A→BC or A→a; used in CYK algorithm
GNF	A→aα; every step consumes one terminal
Pumping Lemma	w=uvxyz; |vxy|≤p, |vy|≥1; uv^n xy^n z ∈ L
CFL ∩ CFL	NOT necessarily CFL (not closed)
CFL ∩ Regular	Always CFL (closed); use PDA×DFA product

Exam Strategy — Key Points to Remember

1. **Chomsky Hierarchy:** Always remember Type 3 ⊂ Type 2 ⊂ Type 1 ⊂ Type 0. Know which automaton corresponds to each type.

2. **Simplification order:** ε-productions → Unit productions → Useless symbols. Changing the order gives wrong results.

3. **CNF conversion order:** Remove ε → Remove units → Remove useless → Fix terminals → Break long RHS.

4. **Pumping Lemma:** Choose w = $a^p b^p c^p$ for 3-symbol languages. Your job is to show ALL splits fail, not just one.

5. **Closure:** CFL is closed under ∪, concat, *, reversal, homomorphism, and ∩ Regular. NOT closed under ∩ (with CFL), complement, or difference.