Context-Free Grammars.

**Grammar notation for recursive definition of language (natural or programming)**

**Example (Palindromes)**

Recall palindromes are strings which read the same forwards and backwards.

Thus \( L_{\text{Pal}} = \{ w \in \{0,1\}^* \mid w^R = w \} \)

**Grammar**

\[
\begin{align*}
P & \rightarrow \varepsilon \\
P & \rightarrow 0 \\
P & \rightarrow 1 \\
P & \rightarrow OP0 \\
P & \rightarrow 1P1
\end{align*}
\]

Basis 0, 0, 1 are palindromes.

Induction if \( P \) is a palindrome then so is \( OP0, 1P1 \).

**Implicit**

- All palindromes can be obtained by applying rules recursively.

- Only palindromes can be so obtained.

**Exercise** Prove all or only by induction on number of rules applied.

**Sample Applications**

\[
\begin{align*}
0110 & \quad P \rightarrow OP0 \rightarrow 01P10 \rightarrow 0110 \\
0010 & \quad P \rightarrow OP0 \rightarrow 00P00 \rightarrow 00100
\end{align*}
\]
EXAMPLE (REGISTRATION LIST IN XML)

SUPPOSE WE WANTED TO GENERATE LISTS OF
REGISTERED STUDENTS IN THE FORMAT:

```xml
<roll>
  <course> CS103 </course>
  <students>
    <name> Sally </name>
    <name> Fred </name>
  </students>
  <course> CS154 </course>
  <students>
    <name> Peter </name>
    <name> Eve </name>
  </students>
</roll>
```

GRAMMAR

```
ROLL → <roll> CLASSES </roll>
{ CLASSES → CLASS CLASSES
   CLASSES → CLASS
   CLASS → CLASS-NAME STUDENT-LIST
   CLASS-NAME → <course> TEXT </course>
   STUDENT-LIST → <students> NAMES </students>
   { NAMES → NAME NAMES
   NAMES → NAME
   NAME → <name> TEXT <name>
   TEXT → CHAR TEXT
   TEXT → CHAR
   CHAR → ... 
   { ALL ASCII CHARACTERS
```
**Example (Natural Language)**

- **Sentence** → **Noun-Phrase** **Verb-Phrase**
- **Noun-Phrase** → **Adjective** **Noun-Phrase**
- **Noun-Phrase** → **Noun**
- **Noun** → boy
- **Noun** → house
- **Adjective** → *little*
- **Adjective** → ...  

**Example (L(\{0,1\})*)**

\[ L_{\text{EQ}} = \{ \omega \in \{0,1\}^* \mid \omega \text{ has equal } # \text{ of } 0\text{s and } 1\text{s} \} \]

**Grammar**

\[
\begin{align*}
S & \rightarrow \epsilon \\
S & \rightarrow OA \\
S & \rightarrow 1B \\
A & \rightarrow 1S \\
A & \rightarrow OAA \\
B & \rightarrow OS \\
B & \rightarrow 1BB \\
\end{align*}
\]

**Idea** "A" generates strings with one more 1 than 0, while "B" generates all strings with one more 0 than 1.

**Verify** "S" generates exactly \( L_{\text{EQ}} \) (Induction).

**Application**

\[
\begin{align*}
S & \rightarrow OA \\
& \rightarrow OOAA \\
& \rightarrow OOISA \\
& \rightarrow 0011S1S \\
& \rightarrow 0011S \\
& \rightarrow 00111B \\
& \rightarrow 001110S \\
& \rightarrow 001110 \\
\end{align*}
\]
FORMALLY  GRAMMAR  \( G = (V, T, P, S) \)

**TERMINALS**  \( T \)  BASICALLY  \( T = \Sigma \)

**VARIABLES**  \( V \)  NON-TERMINAL SYMBOLS THAT
REPRESENT SETS OF STRINGS BEING
DEFINED RECURSIVELY

**START SYMBOLS**  \( S \)  \( \text{EU} \)  IS THE SPECIAL VARIABLE
THAT GENERATES THE DESIRED LANGUAGE.

**PRODUCTION RULES**  \( P \)  RECURSIVE DEFINITIONS.

**NOTE**  \( T, V, \) AND  \( P \)  ARE ALWAYSFINITE SETS

---

**EXAMPLE**  THE GRAMMAR  \( G_{eq} = (V, T, P, S) \)

\[
T = \{ 0, 1 \} \\
V = \{ S, A, B \} \\
P = \{ \\
S \rightarrow \epsilon | 0A | 1B, \\
A \rightarrow 1S | 0AA, \\
B \rightarrow 0S | 1BB \}
\]

---

**NOTATION**  FOR SET OF RULES

\[
A \rightarrow \alpha_1, A \rightarrow \alpha_2, \ldots, A \rightarrow \alpha_k
\]

**SHORT-CUT**

\[
A \rightarrow \alpha_1 | \alpha_2 | \ldots | \alpha_k.
\]
CONTEXT-FREE GRAMMAR (CFG) are only allowed to have production rules for substitution of the form:

\[ A \rightarrow \alpha_1 \alpha_2 \ldots \alpha_R \]

WHERE

L.H.S. \( A \in V \)
R.H.S. \( \alpha_i \in V \cup \Sigma \), for all \( i \)

NON-CONTEXT-FREE? Rules might specify context in which the rule's substitution can be performed.

E.G.: \( O a 1 \rightarrow O \alpha_1 \alpha_2 \ldots \alpha_R \)

Thus rules cannot be applied in other contexts.

DERIVATIONS

SINGLE STEP: For CFG \( G \), \( \forall \alpha, \beta, \gamma \in (\Sigma \cup \Gamma)^* \), \( \forall \epsilon \in \Sigma \)

We say \( \alpha A \beta \Rightarrow \alpha \gamma \beta \) (directly derives)
If the rule \( A \rightarrow \gamma \) is in \( G \).

Means can derive \( \alpha \gamma \beta \) from \( \alpha A \beta \).

DERIVATION SEQUENCE

Suppose \( \alpha_1 \Rightarrow \alpha_2 \Rightarrow \alpha_3 \Rightarrow \ldots \Rightarrow \alpha_{q_1} \)

We say \( \alpha_1 \xRightarrow{*} \alpha_{q_1} \)

Means \( \alpha_1 \) derives \( \alpha_{q_1} \) in 0 or more derivation steps.
**Observation**  
$\Rightarrow$ is reflexive / transitive closure of $\Rightarrow$

- Reflexive: $\alpha \Rightarrow \alpha$
- Transitive: $\alpha \Rightarrow \beta, \beta \Rightarrow \gamma$ implies $\alpha \Rightarrow \gamma$

**Example**

$$G = \{ S \Rightarrow e | OA | IB \}
A \Rightarrow IS | OAA
B \Rightarrow OS | IBB$$

1. $OOIISA \Rightarrow OOIISIS$ (using $A \Rightarrow IS$)
2. $OOIISIS \Rightarrow OOIILS$ (using $S \Rightarrow e$)
3. $OOIISA \Rightarrow OOIILS$ (using $a, b$)
4. $S \Rightarrow 001110$

**Language**

The language of CFG $G = (V, T, P, S)$ is defined as

$L(G) = \{ w \in T^* | S \Rightarrow^* w \}$

**Context-Free Language (CFL)** Any language which has a CFG $G$.

**Terminology**

- **Sentence**: Any $w \in T^*$ such that $S \Rightarrow^* w$ (i.e., well-formed expression).
- **Sentential Form**: Any $\alpha \in (V \cup T)^+$ such that $S \Rightarrow^* \alpha$.

**Terminals** $a, b, c, \ldots$

**Variables** $A, B, C, \ldots$

**Terminal Strings** $\ldots u, v, w, x, y, z$

$VUT \ldots X, Y, Z$

$C \ldots ^*$ forms $d, b, x, \ldots$
**Example (Arithmetic Expressions)**

- $G = (V, T, P, S)$
- $V = \{E, I\}$
- $S = E$
- $T = \{x, y, z, +, *, C, S\}$
- $P$

$$E \rightarrow I \mid (E) \mid E + E \mid E \ast E$$

$$I \rightarrow x \mid y \mid z$$

**Apply**

$$E \Rightarrow E + E \Rightarrow E + E \ast E \Rightarrow I + E \ast E \Rightarrow x + E \ast E \Rightarrow x + I \ast E \Rightarrow x + y \ast E \Rightarrow x + y \ast I \Rightarrow x + y \ast z$$

**Non-Determinism**

**Consider**

$$E \Rightarrow E + E \ast E$$

**Now possible derivation steps?**

- **Expand** any one of three $E$'s
- **Use** any one of four rules for $E$

**Non-Determinism**

**Decide**

- Which variable?
- Which rule?

**Observe**

Deciding which rule to apply is important.

Non-determinism which gives CFG its power & flexibility.

But deciding which variable only serves to confuse us (or, the parser).
**Leftmost Derivation** always substitutes the leftmost variable by a production rule in sentential forms arising in course of a derivation.

**Notation**

\[ \rightarrow_{lm} \]

**Example**

\[ E \rightarrow E + E \rightarrow I + E \rightarrow x + E \]
\[ \rightarrow x + E \rightarrow x + I \rightarrow \ldots \]

**Similarly** can define rightmost derivations

\[ E \rightarrow E + E \rightarrow E + E \rightarrow E + E \rightarrow E + E \rightarrow \ldots \]

**Now** we can talk about "canonical" derivation sequences \( \rightarrow_{lm} \) or \( \rightarrow_{sm} \).

**Derivation Trees**, also called parse trees in compilers, idea represent derivations pictorially.

```
     E
    /|
   / |?
  /  +  \
 /    /\   /
E  /  E   E  E  \\
I /  I   *   I  I
x I   E   E   I
```

Above tree represents the derivation \[ E \xrightarrow{*} x + y * z. \]

In general, to represent \[ A \xrightarrow{*} \alpha \]
- root labeled \( A \)
- leaves in left-to-right order gives \( \alpha \)
- internal nodes are labeled with variables and their children specify the production rule applied to them.

**Example**

![Tree diagram](image)

comes from \[ E \xrightarrow{} E + E \]

A-tree tree rooted at variable \( A \)
- but simply call parse tree if root is \( S \)

Yield/Frontier of a tree is the sequence of leaf labels in left-to-right order.

**Theorem** \( \lambda \) is the yield of an A-tree

\[ \iff A \xrightarrow{\cdot} \lambda \]

**Proof** — induction on tree height
- see textbook
Observe preceding parse tree does not specify a unique way to derive $a$ from $A$.

In fact it removes non-det of which rules to apply, but order is left unspecified.

Leftmost derivation is obtained by traversing the tree in depth-first order, always going into left subtrees before right one.

$$
E \Rightarrow E + E \Rightarrow I + E \Rightarrow x + E \Rightarrow x + E + E
$$

Similarly rightmost derivations come from depth-first traversal going into right subtrees first.

$$
E \Rightarrow E + E \Rightarrow E + E * E \Rightarrow E + E * I \Rightarrow \ldots
$$

**Theorem** Following are all equivalent statements for CFG $G = (V, T, P, S)$ and string $w \in T^*$

1. $w \in L(G)$ or $S \xrightarrow{*} w$
2. $S \xrightarrow{\gamma} w$
3. $S \xrightarrow{\gamma} w$
4. There exists an $S$-tree with yield $w$

**Proof** Obvious

Remark: Here $w$ is the input on a scan.
OF COURSE WE COULD ALWAYS USE $LM$ DERIVATION TO SPECIFY A CANONICAL WAY TO DERIVE ANY $WL(G)$ OR CONVERT PARSE TREE TO UNIQUE DERIVATION — SIMPLIFYING TASK OF PARSERS

BUT WHAT IF SOME $WL(G)$ HAS TWO DISTINCT PARSE TREES & HENCE TWO DISTINCT $LM$ DERIVATIONS?

EXAMPLE $x + y * z$

![Parse Tree Diagram]

**L.H. DERIVATIONS**

**TREE 1**

$E \Rightarrow E + E \Rightarrow I + E \Rightarrow x + E \Rightarrow x + E * E$

**TREE 2**

$E \Rightarrow E * E \Rightarrow E + E * E \Rightarrow ...$
NOTE

This not just a syntactic problem, as we get two different semantic interpretations:

Tree 1: \( x + (y \times z) \)

Tree 2: \((x+y) \times z\).

\[ \text{DEFN A CFG } G \text{ is } \text{ambiguous if for some } w \in L(G) \]

there exist two distinct parse trees.

In compilers, parse tree determine interpretation and we cannot allow ambiguity.

Of course, we can fix problem by forcing use of parenthesis, but we should really re-design the grammar to be unambiguous by encoding precedence of operators (see textbook for re-designed grammar).

While above grammar can be re-designed to remove ambiguity, this is not always possible.

\[ \text{DEFN A CFL is said to be inherently ambiguous if all its grammars are ambiguous} \]

Example:

\[ L = \{ a^n b^m c^{n+m} \mid n, m \geq 1 \} \cup \{ c^n b^m \mid n, m \geq 1 \} \]

Consider strings of the form \( a b b c ) b c d c \).

--- Cannot tell whether this comes from first or second type of strings in \( L \) and any CFG must allow both of these possibilities.