Content Free Language (C.F.L.)
(relevant to programming languages)

\[ G \vdash \text{left side single var, right side special structure} \]

C.F.G. "" "" "" "" "" "" "" "" "" "" "" "" "" "" "" "" "" "" "" "" "" "" "" "" "" "" "" "" "" "" "" "" "" "" "" "" "" "" ""

\[ G = (V, T, S, P) \in \text{C.F.G.} \]

A \rightarrow \alpha

A \in V, \alpha \in (V\cup T)^* \]

C.F.L. is L(C.G.)

Every RG is also CFG, : L(RG) is also L(CFG) \Rightarrow L(CFG) \subseteq L(C.G.)

\[ G: \]

\[ S \rightarrow aSb \]

\[ L(G) = \{a^m b^n : m \geq 0 \} \]

\[ G: \]

\[ S \rightarrow aSb \]

\[ S \rightarrow bSb \]

\[ S \rightarrow \lambda \]

\[ L(G) = \{ w w^R : w \in \{a, b\}^* \} \]

Derivations of Language from Grammar

Take an example of a nonlinear grammar

\[ G: 1) S \rightarrow AB, \ 2) A \rightarrow aAB, \ 3) A \rightarrow \lambda, \ 4) B \rightarrow BB, \ 5) B \rightarrow \lambda \]

Consider a leftmost derivation: leftmost variable is replaced by

\[ S \Rightarrow AB \Rightarrow aAB \Rightarrow aAB \Rightarrow a AB \Rightarrow aab \] (each one is a sentential form)

\[ S \Rightarrow AB \Rightarrow aAB \Rightarrow a AB \Rightarrow aab \]

Derivation Tree:

\[ G: \]

\[ S \rightarrow aAB \]

\[ A \rightarrow bBb \]

\[ B \rightarrow A \lambda \]

Yield: a b b b b (leaves) one example of tree derivation

4) Root is S

5) a \in V, \lambda \notin V

6) \lambda \notin V

Partial Tree

Yield: a b B B

(sentential form)
Let $G = (V, T, S, P)$ be a $CFG$. (a) For every $w \in L(G)$, $\exists$ a derivation tree of $G$ whose yield is $w$. (b) Yield of any derivation tree is in $\mathcal{L}(G)$. (c) If $t_\lambda$ is any partial derivation tree for $G$ with root $S$, then its yield is a sentential form of $G$.

By induction.

Membership: Given $w, G$, find if $w \in \mathcal{L}(G)$.

Parsing: Finding a sequence of productions by which $w \in \mathcal{L}(G)$ is derived.

Exhaustive Search / Top-down Parsing:

- $G$: $S \rightarrow SS | asb | bsa | \lambda$, $w = aabb$
  
  1. $S \rightarrow SS$
  2. $S \rightarrow asb$
  3. $S \rightarrow bsa$
  4. $S \rightarrow \lambda$

  | $s \rightarrow ss \Rightarrow ss$           | $s \rightarrow asb \Rightarrow asbs$ |
  | $s \rightarrow ss \Rightarrow ss$           | $s \rightarrow bsa \Rightarrow bsa$ |
  | $s \rightarrow ss \Rightarrow ss$           | $s \rightarrow \lambda \Rightarrow \lambda$ |

Exhaustive Search / Top-down Parsing:

Problem: (1) $S$ is redundant, (2) It might never end.

Solution: Restrict grammar so that the following type rules are not allowed:

- $A \rightarrow \lambda$
- $A \rightarrow B$

Then parsing will yield a result.

- Each step increases length or number of terminal symbols (both are limited by $|w|$).
- A derivation cannot be more than $2|w|/\lambda$ steps.

The bound on the total # of sentential forms is:

$M = \prod_{i=1}^{P} \prod_{j=1}^{P} ( P_{i,j} - 1 )$, where $P$ are the production rules set.

VCFG:

An algorithm for parsing any $w \in \mathcal{L}(CFG)$ in $\#P$ steps $\propto |w|^3$.

We would like linear-time, i.e. $\propto |w|$

not known for general CFGs

but can be found for restricted ones.
**Defn:** A CFG $=(V, T, S, P)$ is an S-grammar (s.grammar) if $P$ have all rules like

$$A \rightarrow ax$$

$A \in V$, $a \in T$, $x \in V^*$, and any pair $(Aa)$ occurs at most once in $P$.

---

**Ex:** $S \rightarrow as | bss | c$ is an S-grammar

$s \rightarrow as | bss | c$ is not.

Many features of programming languages are S-grammars.

*We L(SG) can be parsed using 3N1 effort.*

**Defn:** Derivation begins with,

$$S \rightarrow a, A, A, \ldots$$

next substitutes for $A$, (only one rule for two) each step produces one terminal symbol, not more than 1W1 steps.

---

**Defn:** A CFG is ambiguous if $w \in L(CFG)$ that has at least two distinct derivation trees.

**Ex:** CFG: $S \rightarrow asb | ss | a$

$w = aabb$ is derivable from

![Diagram](image)

We can try to create an unambiguous form for a CFG.
\[ G = (V, T, \Sigma, \mathcal{P}) \]

\[ V = \{ E, I, S, a, b, c, t, +, *, (, ) \} \]

\[ \mathcal{P} = 
  \begin{align*}
  &E \rightarrow I, E \rightarrow E + E, E \rightarrow E \times E, E \rightarrow (E), I \rightarrow a1b1c
\end{align*}
\]

(Arithmetic expression in C-like languages)

The grammar is ambiguous.

\[ E \rightarrow a + b \times c \] has two derivations:

\[ \begin{align*}
  E &\rightarrow E + E \\
  &\rightarrow E + E + E \\
  &\rightarrow (E) + E \\
  &\rightarrow (E) + (E) + E
\end{align*} \]

Change the grammar to:

\[ G' = E \rightarrow T ; T \rightarrow F, F \rightarrow I, E \rightarrow E + T, T \rightarrow T + F, F \rightarrow (E), I \rightarrow a1b1c \]

The grammar is now unambiguous.

\[ G \sim G' \] are equivalent.

Here ambiguity is grammar.

Define: If \( L \in \text{CFL} \) has at least one unambiguous grammar then \( L \) is unambiguous language. If every \( G(L) \) is ambiguous then \( L \) is ambiguous.

\[ L = \{ a^n b^n c^m \} \cup \{ a^0 b^0 c^0 \} \] is ambiguous.

\[ L = L_1 \cup L_2 \]

\[ \begin{align*}
  &\text{CFG}(L_1) : \quad S \rightarrow S_1 \quad \text{CIA} \\
  &\quad A \rightarrow a A b 1 \lambda \\
  &\text{CFG}(L_2) : \quad S_2 \rightarrow a S_1 b 1 \lambda \\
  &\quad B \rightarrow b B c 1 \lambda \\
  &\text{CFG}(L) : \quad S \rightarrow S_1 \mid S_2 \\
  &\text{study } a^n b^n c^n \text{ for } L \text{ derivation.} \]
Programming Languages

Regular languages for expressions are legal CFL for some other aspects, e.g. matching ( and ).

Grammar for programming language in Backus-Naur form (BNF).

E ::= T | E + T
T ::= F | T * F

This arrow G' would have

E ::= <expression> | <term> + <expression>,
<expression> ::= <term> | <expression> * <term>,
<term> ::= <factor> | <term> * <factor>,
<factor> ::= <variable> | variable, +, * are terminals same as before.

In compilers, LL, LR grammars are used.

Ambiguity should be avoided.

Generally, CFG check syntax (and not generally semantics).

If int x = 2.2, has 2 statements correct in syntax but not in semantics.