CFG: context-free grammar

Context-free language

Finite terminal alphabet of tokens \( \Sigma \)
(lower-case word, like then)

Finite non-terminal alphabet \( N \)
like "variables" : (initial-op)

Start symbol \( S \in N \)

Productions \( A \rightarrow X_1 \ldots X_m \)
\( \downarrow \) non-terminal \( \in N \cup \Sigma \)

Allow / (alternative)

\( A \rightarrow ab \) \| \( A \rightarrow ab \) cd

Derivations: \( S \Rightarrow \cdots \Rightarrow \)

(only terminals)

\( S \xrightarrow{*} \)

Conventions: Leftmost derivation: always expand the leftmost non-terminal in the current sentential form.
Rightmost derivation expands rightmost non-terminal = canonical derivation. Bottom-up parsing parses discovered in reverse order.

Parse tree: describes a derivation

![Parse Tree Diagram]

Terminals: [a + b = ...]

Both leftmost, rightmost derivations yield same parse tree. Phrase (all terminals descending from a non-terminal)

Complexity of parsing in tokens: $\mathcal{O}(n^3)$ Useful CFGs (actual ones for programming langs) can be parsed in $\mathcal{O}(n)$.

Unreduced grammar: useless non-terminals: never generated by any sentence.

\[
S \rightarrow A \\
A \rightarrow a \\
B \rightarrow B b \\
C \rightarrow a
\]
Ambiguous grammar: a single sentence has more than one parse tree. Doesn’t naturally occur. Unavoidable for it.

\[ E \rightarrow E - E \]  \[ \rightarrow \text{id} \]

Sentence: \( a - b - c \)

BNF (Backus-Naur Form for writing CFG)

metacharacters

\[ [ \ ] \] to surround optional parts.
\[ \{ \} \] to surround parts with implicit Kleene \(*\).

easily converted to ordinary BNF:
\[ A \rightarrow B \{c\} \quad \| \quad A \rightarrow B \]
\[ A \rightarrow BC \]
\[ A \rightarrow B \epsilon C D \quad | \quad A \rightarrow B X D \]
\[ X \rightarrow \gamma \]
\[ X \rightarrow C X \]

Parser vs. Recognizer vs. Generator
L⇒ builds a tree
⇒ returns Boolean
⇒ creates a valid sentence.
Bottom-up vs Top-down parsers

LR(1)
⇒ rightmost

LL(1)
⇒ input in order
⇒ leftmost parse

LL(1): recursive descent

Analyze the BNF:
Empty: Rule Derives Empty (p)
Symbol Derives Empty (N)
First (N)
Follow (N)
Predict (p)
Empty check

1) Make a list $L$ of Ns that directly derive $\lambda$.

2) For each $N \in L$, for all rules where $N$ is in the RHS, remove $N$ from RHS, if RHS is now empty, put LHS into $L$.

(Use a count of RHS elements to make this step easy)

3) $L$ is the set of Ns that derive $\lambda$. The rules that cause us to put all into $L$ are the rules that can derive $\lambda$.

Programming hint: Avoid duplicates in $L$.

First ($\alpha$) = \{ $b \in \Sigma$ | $\alpha \Rightarrow^* b\beta$ \}

String of non-terminals and terminals

Hint: If BNF is written top-down: work from the end to the beginning.

Method: 1) Consider first element $x$ of $\alpha$.
   - if $\alpha = \lambda$, answer is $\emptyset$.
   - if $x \in T$ (terminal), answer is $\exists \xi$.

2) if $x \in N$,
   - for all productions with $x$ as the LHS, recursively compute First(\text{RHS}),
   - answer = union of all these computations.
### 0. Deriving Empty String

If symbol derives \( E \), compute \( \text{First}(\text{remainder of } \alpha) \)

* \( \alpha \) removing \( E \) at start

Answer = union of that as well as answer

Derives empty: \( \{ E, T \} \)

\[
\begin{align*}
\text{First}(E) &= \{ \varepsilon, f \} \\
\text{First}(T) &= \{ \varepsilon, f \}
\end{align*}
\]

\[
\begin{align*}
\text{First}(\varepsilon) &= \{ \varepsilon \} \\
\text{First}(\varepsilon + E) &= \{ \varepsilon, f \} \\
\text{First}(\varepsilon, T) &= \{ \varepsilon, f \}
\end{align*}
\]

### Follow(N)

Follow(\( N \)) = set of terminals that can come directly after \( N \) in a sentential form.

\[
\{ b \in \Sigma \mid \exists \alpha N b \beta \}
\]

Start non-terminal

Strings of non-terminals and terminals

Method: Follow(\( N \))

For each place \( N \) appears in a RHS of production \( P \)

Answer \( U = \text{First}(\text{tail}) \)

\( \text{if tail} \Rightarrow \varepsilon \)

Answer \( U = \text{Follow}(\text{RHS}(P)) \)