CS 345
Programming Languages

2: Syntax – From Grammars to Languages to ASTs

Motivation

- **What is Syntax?**
  - Noam Chomsky: “The study of principles and processes by which sentences are constructed in particular languages”

- **Why is syntax important for programming languages?**
  - Human factor
  - Machine factor

Alphabet and Strings

- **Def**: An Alphabet \( \Sigma \) is a finite, non-empty set of symbols
- **Examples**: 
  - \( \Sigma_1 = \{0, 1\} \)
  - \( \Sigma_2 = \{\text{true, false}\} \)
  - \( \Sigma_3 = \{0, 1, 2, 3, 4, 5, 6, 7, 8, 9, A, B, C, D, E, F\} \)
  - \( \Sigma_4 = \{❤, ♤, ★, ☀\} \)

- **Def**: A String \( s \) is a finite sequence of symbols in \( \Sigma \).
- **Examples**: 
  - \( s_1 = 010010001 \) from \( \Sigma_1 \)
  - \( s_2 = ❤️♠️★ \) from \( \Sigma_4 \)
  - \( s_3 = \epsilon \) (empty string)

Strings are generated by concatenating symbols

Kleene Star

- **\( k \)-th power**
  - **Def (formal)**: \( \Sigma^0 := \{\epsilon\}, \Sigma^1 := \Sigma, \Sigma^{k+1} := \{ab : a \in \Sigma^k, b \in \Sigma\} \) for \( k > 0 \)
  - **Def (intuitive)**: The \( k \)-th power of \( \Sigma \) is the set of all possible strings \( s_i \) in \( \Sigma \) with \( |s_i| = k \).
  - **Example (for alphabet \( \Sigma = \{0, 1\})**: 
    - \( \Sigma^0 = \{\epsilon\}, \Sigma^1 = \{0, 1\}, \Sigma^2 = \{00, 01, 10, 11\}, ... \)

- **Kleene Operator (Kleene Star, Kleene Closure)**
  - \( \Sigma^* := \Sigma^0 \cup \Sigma^1 \cup \Sigma^2 \ldots = \bigcup_{i=0}^{\infty} \Sigma^i \)
  - \( \Sigma^+ := \bigcup_{i=1}^{\infty} \Sigma^i = \Sigma^* \setminus \{\epsilon\} \)
Language

- **Def:** $L \subseteq \Sigma^*$ is called a Language over Alphabet $\Sigma$.
  - Examples:
    - $L_1 = \{0,1,101,100\}$ is a Language over $\Sigma$.
    - $L_2 = \{1,10,100,\ldots\}$ is a Language over $\Sigma$.
  - A Language describes all strings (words) contained in the Language.

- **Languages are Sets**
  - Standard set operations like union, intersection, complement are applicable to Languages.

Grammar

- A Grammar $G$ is defined by the Tuple $G = (V, \Sigma, S, P)$ where
  - $V$ is an alphabet of non-terminal symbols ("variables").
  - $\Sigma$ is an alphabet of terminal symbols
  - $S \in V$ is a start symbol
  - $P$ is an unordered set of productions (relations) of the form $A \rightarrow B$ where $A \in (V \cup \Sigma)^*$, $B \in (V \cup \Sigma)^*$

Grammar Example

- $G_1 = \{(S,T,O,I),\{0,1\},S,P\}$ where $P$ contains
  - $S \rightarrow OT$
  - $S \rightarrow OI$
  - $T \rightarrow SI$
  - $O \rightarrow 0$
  - $I \rightarrow 1$

Describes the set $\{0^n1^n\mid n \geq 1\}$

Sentential Form and Direct Derivation

- **Def:** A Sentential Form $\gamma$ of $G$ is any string of terminals and non-terminals in $G$, $\gamma \in (\Sigma \cup V)^*$

- **Def:** Let $\gamma_1$ and $\gamma_2$ be sentential forms of $G$.
  - $\gamma_1 \Rightarrow \gamma_2$ (\(\gamma_1\) directly derives $\gamma_2$) iff
  - $\gamma_1 = \sigma\alpha\tau$ and $\gamma_2 = \sigma\beta\tau$ and $\alpha \rightarrow \beta$ is a production in $G$.

- **Example:** $00S11 \Rightarrow 00OI11$ in $G_1$

- **Intuition:** $\gamma_1 \Rightarrow \gamma_2$ if $\gamma_2$ is formed by applying a single production to $\gamma_1$. 
Derivation

- Def: The Derivation “⇒∗” is the transitive reflexive closure of the Direct Derivation “⇒”
- Def: Let $\gamma_1$ and $\gamma_2$ be sentential forms of $G$. $\gamma_1 \Rightarrow^* \gamma_2$ (if there exists a sequence of zero or more sentential forms $\sigma_1, ..., \sigma_n$ such that $\gamma_1 \Rightarrow \sigma_1 \Rightarrow ... \Rightarrow \sigma_n \Rightarrow \gamma_2$
(a Derivation of $\gamma_2$ from $\gamma_1$).

Generating Languages from Grammars

- Let $G = (V, \Sigma, S, P)$ be a Grammar.
- Def: The Language $L$ generated by $G$ (“$L(G)$”) is defined as \{ $s : s \in \Sigma^* \land S \Rightarrow^* s$ \}
- Strings $s \in L$ are called Sentences.

Strings $s \in L$ are called Sentences.

- $000111$ is in $L(G_1)$
- $000111$ consists only of terminal symbols
- $S \Rightarrow OT \Rightarrow OSI \Rightarrow OOTI \Rightarrow OOSII \Rightarrow 0001III \Rightarrow ... \Rightarrow 000111$ is one possible derivation from $S$.

Lexical Analysis

- In practice, the terminal symbols are non-trivial.
- Often described as regular expressions (patterns).
  - E.g., digit : [0-9]
- Lexers identify lexemes in the character strings and form tokens to pass them on to the parser.
- Lexers can be hand-coded or generated either by dedicated tools (e.g., Flex) or integrated lexing/parsing tools (e.g., JavaCC, ANTLR).

Chomsky Hierarchy

- Introduced by Noam Chomsky in 1956.
- Based on restrictions.
- Let $G = (V, \Sigma, S, P)$ be a Grammar.
- Def: $G$ is a Type 0 or Unrestricted Grammar.

- For every language generated by a Type 0 grammar there exists a Turing Machine which accepts the language and vice versa.
Context-Sensitive Grammars

- If each Production $\alpha \rightarrow \beta$ satisfies $|\alpha| \leq |\beta|$, then $G$ is a Type 1 or Context-Sensitive Grammar.
  - The production $S \rightarrow \epsilon$ is allowed if $S$ never appears on a right-hand side of any production.

- Equivalent definition: all Productions must be of the form: $\alpha \gamma \rightarrow a \beta \gamma$ with $\alpha, \gamma \in (\Sigma \cup V)^*, \beta \in (\Sigma \cup V)^+, A \in V$
  - (and again $S \rightarrow \epsilon$ allowed if $S$ never appears on any RHS).

- Context-sensitive languages can be decided by a linear-bounded non-deterministic automaton
  - (and, vice versa, every formal language that can be decided by such an automaton is context-sensitive).

Context-Free Grammars

- If each Production in $P$ is of the form $A \rightarrow \beta$ with $A \in V$ and $\beta \in (\Sigma \cup V)^*$, then the grammar is a Type 2 or Context-Free grammar.

- Context-Free languages can be decided by a non-deterministic pushdown automaton (and vice versa).

- Context-Free grammars are fundamental to compiler construction.

Regular Grammars

- If the productions in $P$ are in one of the three forms $A \rightarrow b, A \rightarrow bC, A \rightarrow \epsilon$ with $A, C \in V, b \in \Sigma$, then the grammar is a Type 3 or Regular Grammar.
  - This is a right-regular grammar
  - Left-regular: $A \rightarrowCb$

- For every language generated by a right-regular grammar, there is an equivalent left-regular grammar generating the same language.

- Regular languages can be decided by a finite automaton.

Application: Regular Expressions

- Regular language to describe/match strings.
  - Many meta-characters like `|` for choice, `?` for zero or one elements, `*` for zero or more, `+` for one or more, ...
  - Character classes like `\w` for words, `\d` for single digit, ...

- Example: `^[\w\-\.]+@([\w\-\.]+)[\w\-\{2,4\}]$`

- Regex are often used to make grammars even more compact
Parsing Formal Languages

- Membership problem: Given a string $s$ over $\Sigma$, does it belong to $L(G)$.
  - Type 0 Languages: undecidable (or semi-decidable)
  - Type 1 Languages: decidable
  - Type 2 Languages: decidable in polynomial time
  - Type 3 Languages: decidable in linear time

- Parsing problem: Given a string $s'$ in $L(G)$, how can it be derived from $S$.

Complexity

A Language for which no Grammar Exists

<table>
<thead>
<tr>
<th>$w_1$</th>
<th>$w_2$</th>
<th>$w_3$</th>
<th>$w_4$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$G_1$</td>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>$G_2$</td>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>$G_3$</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>$G_4$</td>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
</tbody>
</table>

The Language $L_d$ is the complement of the diagonal.

If $L_d$ was described by a grammar $G_j$, then would $w_j$ be in $L_d$?

⇒ there is no grammar which describes $L_d$ ⇔ $L_d$ is not recursively enumerable
⇔ there is no TM which accepts $L_d$.

Pumping Lemma

- We will see that the type of a language constraints how we can efficiently parse it
- It is sometimes necessary to proof a language to not be of a certain type.
- The Pumping Lemma can be used to proof a language to be non-regular.
- There is also a Pumping Lemma for context-free languages.
Let $G = \{(0, 9, +, \cdot), \{S, T, F\}, S, P\}$ where $P$ is:

$S \rightarrow T + S, \quad S \rightarrow S + T, \quad S \rightarrow T$

$T \rightarrow F * T, \quad T \rightarrow T + F, \quad T \rightarrow F$

$F \rightarrow <\text{number}>, \quad F \rightarrow (S)$

Parse Tree for $1 + 2 * 3 + 4$:

- $T = 1, S = 2 * 3 + 4$
- $T = 2 * 3, S = 4$
- $F = 2, T = 3$

This is another valid parse tree for $1 + 2 * 3 + 4$:

- $S = 1 + 2 * 3, T = 4$
- $S = 1, T = 2 * 3$
- $F = 2, T = 3$

The grammar is ambiguous!

Ambiguity exists.

Natural language example: Fruit flies like apples.

Some CF grammars are inherently ambiguous, i.e., they cannot be expressed by an unambiguous grammar.

The question whether a given CFG is ambiguous or not is undecidable.

In many programming languages, ambiguity is mitigated by side-conditions like precedence, associativity, etc.

C → if $a$ then $b$

C → if $a$ then $b$ else $c$

Parse: if $A$ then if $B$ then $C$ else $D$

Some compilers group the “else” with the nearest “if”

Some languages prevent the Dangling Else Problem through their grammar
Parsing

- Finding a derivation for a given string efficiently is involved.

- Cocke, Younger, Kasami presented an algorithm (CYK algorithm) that parses CFLs in Chomsky Normal Form in $O(n^3)$ time through dynamic programming.

Chomsky Normal Form

- A grammar is in Chomsky Normal Form if all productions are of the form $X \rightarrow YZ$ or $X \rightarrow a$ where $X, Y, Z \in V$, $a \in \Sigma$

- Theorem: Every CFG $G$ can be transformed into a Chomsky Normal Form $G'$ such that $L(G') = L(G) \setminus \{\varepsilon\}$

CYK Algorithm

Grammar:

<table>
<thead>
<tr>
<th>0</th>
<th>1</th>
<th>2</th>
</tr>
</thead>
<tbody>
<tr>
<td>$O \rightarrow +$</td>
<td>$S \rightarrow d$</td>
<td>$S \rightarrow d$</td>
</tr>
<tr>
<td>$F \rightarrow SS$</td>
<td></td>
<td></td>
</tr>
<tr>
<td>$O \rightarrow -$</td>
<td></td>
<td></td>
</tr>
<tr>
<td>$S \rightarrow d$</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

$O(n^3)$ is unfortunately not practical since input files can be large. In 2013 the Linux kernel had about 2M characters of code excluding any device drivers.

Assuming an average token length of 10, this gives $T(8.000.000.000.000.000)$.

Exorciser

- Demo
Parsing

- **Bottom-Up Parser**
  - Start with the terminals, try to derive the start symbol.
  - CYK is bottom-up.
  - Popular approach: LR(1)
    - Left-to-Right, Rightmost derivation, finite look-ahead of 1

- **Top-Down Parser**
  - Start with the start symbol, try to derive the given string from it. Example: Recursive Descent
  - Simple approach: Backtracking
  - Popular approach: LL(1)
    - Left-to-Right, Leftmost derivation, look-ahead of 1

---

**Predictive Parsing**

- Idea: to prevent backtracking, we want to be able to distinguish in cases where the grammar provides choices. This is called **Predictive Parsing**

  - The First set:
    - Def: \( \text{First}(A) = \{ b | A \Rightarrow^* b\alpha \} \cup \{ \epsilon \text{ if } A \Rightarrow^* \epsilon \} \)
      \[ A \in V, \quad b \in \Sigma, \quad \alpha \in (V \cup \Sigma)^* \]

---

**Example:**

**Grammar:**

- \( S \rightarrow TX \)
- \( T \rightarrow (S) \)
- \( T \rightarrow \text{int}\ Y \)
- \( X \rightarrow +S \)
- \( X \rightarrow \epsilon \)
- \( Y \rightarrow +T \)
- \( Y \rightarrow \epsilon \)

**First Set Calculations:**

- \( \text{First}(T') = \{ \star, \epsilon \} \)
- \( \text{First}(X) = \{ \star, \text{int}\} \)
- \( \text{First}(T) = \{ \text{int}\} \)
- \( \text{First}(S) = \text{First}(T) = \{ \text{int}\} \)
- \( \text{First}(T) = \{ \text{int}\} \)
- \( T \rightarrow \text{int}\ Y \)
- \( Y \rightarrow +T \)
- \( Y \rightarrow \epsilon \)

---

**Creating the LL(1) Parsing Table**

- for each terminal \( \alpha \in \text{First}(A) \) do
  - add \( A \rightarrow \alpha \) to \( T[A, \alpha] \).

<table>
<thead>
<tr>
<th>( S \rightarrow TX )</th>
<th>( T \rightarrow (S) )</th>
<th>( T \rightarrow \text{int}\ Y )</th>
<th>( X \rightarrow +S )</th>
<th>( X \rightarrow \epsilon )</th>
<th>( Y \rightarrow +T )</th>
<th>( Y \rightarrow \epsilon )</th>
</tr>
</thead>
<tbody>
<tr>
<td>( \text{int}\ \star\ +\ (\ )\ S )</td>
<td>( \text{int}\ \star\ +\ (\ )\ S )</td>
<td>( \text{int}\ \star\ +\ (\ )\ S )</td>
<td>( \text{int}\ \star\ +\ (\ )\ S )</td>
<td>( \text{int}\ \star\ +\ (\ )\ S )</td>
<td>( \text{int}\ \star\ +\ (\ )\ S )</td>
<td>( \text{int}\ \star\ +\ (\ )\ S )</td>
</tr>
<tr>
<td>( S \rightarrow TX )</td>
<td>( T \rightarrow (S) )</td>
<td>( T \rightarrow \text{int}\ Y )</td>
<td>( X \rightarrow +S )</td>
<td>( X \rightarrow \epsilon )</td>
<td>( Y \rightarrow +T )</td>
<td>( Y \rightarrow \epsilon )</td>
</tr>
</tbody>
</table>
Using the LL(1) Parsing Table

**Parsing:**
- int + int
- S → TX
- T → ( S )
- T → int Y
- Y → ε

**Stack**

<p>| | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>S</td>
<td>TX</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>T</td>
<td>int Y</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>X</td>
<td>+ S</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Y</td>
<td>$</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**Problem:**

Predictive Parsing

- Problem: When we have ε in the first set we cannot make a decision based on the current symbol alone
- Solution: we need to look ahead

**The Follow set:**

- Def: \( \text{Follow}(A) = \{ b | S \Rightarrow^* aAb \} \cup \{ \$ \text{ if } \exists \gamma \in (V \cup \Sigma)^* : S \Rightarrow^* \gamma A \} \)
  \( A \in V, \ b \in \Sigma, \ a, \beta \in (V \cup \Sigma)^* \)

Now, we can decide whether we need an ε replacement.
- We can check if the current symbol is in \( \text{Follow}(A) \)

Creating the LL(1) Parsing Table

- for each terminal \( a \in \text{First}(A) \) do
  - add \( A \rightarrow a \) to \( T[A,a] \).
- if \( \epsilon \in \text{First}(A) \) then
  - for each terminal (or delimiter) \( a \in \text{Follow}(A) \) do
    - add \( A \rightarrow a \) to \( T[A,a] \).
    - (i.e., add \( \epsilon \))

<p>| | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>S</td>
<td>TX</td>
<td>TX</td>
<td></td>
<td></td>
</tr>
<tr>
<td>T</td>
<td>int Y</td>
<td>$</td>
<td></td>
<td></td>
</tr>
<tr>
<td>X</td>
<td>+ S</td>
<td>$</td>
<td>$</td>
<td>$</td>
</tr>
<tr>
<td>Y</td>
<td>$</td>
<td>$</td>
<td>$</td>
<td>$</td>
</tr>
</tbody>
</table>
Using the LL(1) Parsing Table

 Parsing:
- int + int
- S -> TX
- T -> int Y
- Y -> e
- X -> +S
- S -> TX
- T -> int Y
- Y -> e

<table>
<thead>
<tr>
<th>Stack</th>
<th>Action</th>
</tr>
</thead>
<tbody>
<tr>
<td>S</td>
<td>TX</td>
</tr>
<tr>
<td>T</td>
<td>int Y</td>
</tr>
<tr>
<td>X</td>
<td>+S</td>
</tr>
<tr>
<td>Y</td>
<td>eT</td>
</tr>
</tbody>
</table>

LL(1) Parsing

- A context-free grammar which has no multiply-defined entries in its parsing table is called an LL(1) grammar.
- Reasons for grammars not being in LL(1):
  - Grammar is left-recursive
  - Grammar is not left-factored
  - Grammar is ambiguous
- Many existing languages are not expressible through a LL(1) grammar

LR Parsing

- Bottom-up parsing
- LR Parsers
  - work for almost all PL constructs
  - can be implemented very efficiently
  - are more powerful than LL Parsers
  - detect errors early
  - are unfortunately involved to construct => typically done by automated tools
- For details on how to construct LR parsers, consult the Dragonbook.

Informational: LR Parsing Example

<table>
<thead>
<tr>
<th>State</th>
<th>int</th>
<th>*</th>
<th>+</th>
<th>...</th>
<th>$</th>
<th>int</th>
<th>*</th>
<th>+</th>
<th>...</th>
<th>$</th>
<th>S</th>
<th>...</th>
<th>Y</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>in</td>
<td></td>
<td></td>
<td></td>
<td>$</td>
<td>1</td>
<td></td>
<td></td>
<td></td>
<td>$</td>
<td>3</td>
<td></td>
<td>8</td>
</tr>
<tr>
<td>1</td>
<td>p2</td>
<td>accept</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>in</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>3</td>
</tr>
<tr>
<td>...</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th></th>
<th>ACTION</th>
<th>GOTO</th>
</tr>
</thead>
<tbody>
<tr>
<td>*(+ int) int</td>
<td>Stack</td>
<td>Input</td>
</tr>
<tr>
<td>0</td>
<td>*(+ int) int</td>
<td>shift</td>
</tr>
<tr>
<td>1*0</td>
<td>(+ int) int</td>
<td>...</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
<td>...</td>
</tr>
</tbody>
</table>

Ambiguous grammars create (shift/reduce, shift/shift) conflicts.
### BNF

- Backus-Naur Form is a widely used form of expressing grammars.
- Originally: `<symbol> ::= expression`
  - non-terminals always in brackets
  - The vertical bar `|` indicates a choice (compacts multiple predicates for given non-terminal)

- Extended Backus-Naur Form (EBNF) uses `=` for definition and adds additional meta-characters like `[ ... ]` for option, `{ ... }` for repetition, ...

### Parse Trees and ASTs

- Parse tree contains unnecessary information
- The Abstract Syntax Tree is more compact
- Parsers usually generate ASTs

#### Parse Tree

```
    5
   / \
 4   6
 / \ /  \
2  3 T  T
  1 + T
```

#### AST

```
+ 1 4
  + 2 3
```

### Pumping Lemma

- Let $L$ be a language and $s \in L$ a word in the language with $|s| \geq p$.
- If $L$ is regular, then there exists a number $p$ so that a decomposition of $s: s = xyz$ exists such that
  $$|y| > 0, \quad |xy| \leq p, \quad \forall k \geq 0: xy^k z \in L.$$  
  $p$ is called the pumping length.

- Intuition: The membership problem for regular languages can be decided by a FSA. If the input string length exceeds the number of states and is still accepted by the FSA, the automaton must pass a state at least twice (pigeon-hole principle). We can exploit this by “pumping” the string.
Pumping Lemma Applied

\[ L = \{aa, abaa, ababaa, abababaa, \ldots \} \]
\[ L = (ab)^*aa \]
\[ p = 3 \]
\[ \begin{array}{ccc}
    & a & \\
    \text{Start} & b & a \\
\end{array} \]
\[ y \neq \epsilon, \quad |xy| \leq p \]
\[ \forall k \geq 0 : xy^kz \in L \]
\[ y^0z = \{aa\} \in L, y^1z = \{bab\} \in L, y^2z = \{ababaa\} \in L, \ldots \]

Pumping Lemma for CF Languages

\[ 0^n1^n = xyz \text{ where} \]
\[ y \neq \epsilon, \quad |xy| \leq n, \quad \forall k \geq 0 : xy^kz \in L \]
\[ y \neq \epsilon \Rightarrow |x| < n \]
\[ x0y^0z \text{ cannot be } \in L \text{ since it contains fewer } 0s \text{ than } 1s \Rightarrow \]
\[ L \text{ cannot be regular} \]

Pumping Lemma Applied

\[ L = \{0^n1^n | n \geq 1\}. \text{ Is } L \text{ regular?} \]
\[ \text{Pumping Lemma: If } L \text{ was regular, we could write } 0^n1^n = xyz \text{ where} \]
\[ y \neq \epsilon, \quad |xy| \leq n, \quad \forall k \geq 0 : xy^kz \in L \]
\[ (\text{we chose } p = n) \]
\[ |xy| \leq n \Rightarrow x \text{ and } y \text{ consist of } 0s \text{ only.} \]
\[ z \text{ contains } n \text{ } 1s. \]
\[ y \neq \epsilon \Rightarrow |x| < n \]
\[ xy^0z \text{ cannot be } \in L \text{ since it contains fewer } 0s \text{ than } 1s \Rightarrow \]
\[ L \text{ cannot be regular} \]

Pumping Lemma for CF Languages

\[ \text{Let } L \text{ be a language and } s \in L \text{ a word in the language with } |s| \geq p. \]
\[ \text{If } L \text{ is context-free, then there exists a number } p \text{ so that a decomposition of } s \text{ of the form } s = uvwxy \text{ exists such that} \]
\[ vx \neq \epsilon, \quad |vwxy| \leq p, \quad uv^kwx^ky \in L \forall k \geq 0 \]

\[ \text{Intuition: CF grammars correspond to PDAs which can count arbitrarily high but the LIFO access implies that} \]
\[ \text{they can use their stack only to count a single counter.} \]
**Constructing the CNF**

1. **Add a new start variable** $S_0$ and the rule $S_0 \rightarrow S$

2. **Eliminate the $\epsilon$ rules** ($X \rightarrow \epsilon$)
   
   If we have rules where $\epsilon$ appears on the RHS, eliminate these rules by copying every rule on which the LHS appears and applying the $\epsilon$ rule to the copies. The $\epsilon$ rule can consequently be eliminated.

3. **Eliminate the unit rules**
   
   If we have rules of the form $A \rightarrow B$, then wherever the rule $B \rightarrow u$ appears, add the rule $A \rightarrow u$ unless this rule was already replaced.

4. **Clean up rules with two or more symbols**
   
   Introduce new variables to break up rules of the form $A \rightarrow u_1 u_2 u_3 \ldots u_n$ ($n > 2$) into a set of rules $A \rightarrow u_1 A_1, \ A_1 \rightarrow u_2 A_2, \ldots, A_{n-2} \rightarrow u_{k-1} u_k$.

   Now all rules have at most two symbols on the RHS. For those having a terminal as one of the two symbols, substitute the terminal with a newly introduced variable:
   
   $A \rightarrow uB$ becomes $A \rightarrow UB, \ U \rightarrow u$.

   ▶ The resulting grammar is in Chomsky Normal Form.