CONTEXT FREE GRAMMARS/LANGUAGES

and PUSH DOWN AUTOMATA

We'll now study the class of "context free languages" (CFLs). This class
- includes the class of regular languages,
- is characterized by "context free grammars" (CFGs) in syntactic manner,
- is characterized (equivalently) by "push down automata" (PDAs) in computational manner.

We start with grammars (CFGs).

Motivating example from natural language (English)

Sentence $\rightarrow$ Subject Verb Object
Sentence $\rightarrow$ Sentence and Sentence
Subject $\rightarrow$ Jim | Lisa
Verb $\rightarrow$ ate | threw | killed
Object $\rightarrow$ apple | ball | rat

This grammar generates sentences such as
- Jim ate apple
- Jim ate apple and Lisa threw ball
Lisa threw apple and Jim killed rat and Jim ate ball

Fact: Programming Languages (such as C) are described by a grammar that generates all syntactically correct programs.

Exercise: Look up the C grammar.

Example of a CFG

\[ G: \begin{align*}
S &\rightarrow A \\
A &\rightarrow 0A1 \\
A &\rightarrow \#
\end{align*} \]

- 3 "rules"
- S, A variables
- S: start variable
- 0, 1, # terminals

Idea: Start with the string \( S \).
At any step, replace a variable \( \alpha \) by string \( \gamma \) if \( \alpha \rightarrow \gamma \) is a rule.
Stop if the string only has terminals and no variable.

Eg: \( S \rightarrow A \rightarrow OA1 \rightarrow 0OA11 \rightarrow 00\#11 \)

We say that grammar \( G \) derives string \( 00\#11 \).
Clearly this grammar generates precisely the set of strings (referred to as the language \( L(G) \) generated by the grammar)

\[ L(G) = \{ 0^n \# 1^n \mid n \geq 0 \} \]

Note that \( L(G) \) is not regular.

**Parse tree**

The parse tree shows graphical (and structural) representation of the derivation. (Of string 000\#111 here)

Formal Definition of CFG

**Def** A CFG \( G \) is a 4-tuple \( G = (V, \Sigma, R, S) \) where

- \( V \) is a finite set of variables
- \( \Sigma \) is a set of terminals, disjoint from \( V \)
- \( S \in V \) is the start variable.
- $Q$ is a finite set of rules of type
Variable $\rightarrow$ String of variables and terminals
i.e. $A \rightarrow (\mathcal{V} \cup \Sigma)^*$, $A \in Q$.

Usually the start variable appears as the left side of the topmost/first rule.

**Def** Suppose $u, v, w \in (\mathcal{V} \cup \Sigma)^*$ and
$A \rightarrow w$ is a rule. Then we say that
$uAv \Rightarrow uvw$ "$uAv$ yields $uvw$".
I.e. one substitutes $A$ by the right side of the rule $w$.

**Def** $u \Rightarrow^* v$ if there is a (finite)
sequence of strings $u_1, \ldots, u_k \in (\mathcal{V} \cup \Sigma)^*$
such that
$u_1 = u \Rightarrow u_1 \Rightarrow u_2 \Rightarrow \ldots \Rightarrow u_k = v$.

**Def** Language generated by grammar $G$ is
$L(G) = \{ w \in \Sigma^* \mid S \Rightarrow^* w \}$.
Note. \( L(G) \) is (only) over the alphabet \( \Sigma \) of terminals.

Example

\[
G: \quad S \to (S) \\
S \to SS \\
S \to \varepsilon
\]

\[\begin{array}{c}
S \to (S) \\
\quad \to (SS) \\
\quad \to ((S)S) \\
\quad \to ((S)(S)) \\
\quad \to ((S)(S)) \rightarrow (XX)
\end{array}\]

\( L(G) \) is the language of all "matched parentheses". The substituted variable is underlined in each step above.

Example

\[
G_{\text{Add}}: \quad E \to E + E \\
E \to 1 | 2 | 3
\]

\( L(G_{\text{Add}}) \) is all expressions of type

\[1, 1+2, 1+2+3, 2+1+1+3+2, \text{ etc.}\]
Example

\[ G_{\text{Add-Mult}} : \]
\[
E \rightarrow E + E \quad \Sigma = \{+, \times, 1, 2, 3\}
\]
\[
E \rightarrow E \times E
\]
\[
E \rightarrow 1 \mid 2 \mid 3
\]

\( L(G_{\text{Add-Mult}}) \) has arithmetic expressions such as
\[
1 + 2 \times 3 \quad 2 + 3 \times 1 + 2 \quad \text{etc.}
\]

\[
E \rightarrow E \times E \rightarrow E + E \times E \rightarrow 1 + E \times E
\]
\[
\rightarrow 1 + E \times 3 \rightarrow 1 + 2 \times 3
\]

Example

\[ G_{\text{Arith}} : \]
\[
E \rightarrow (E + E) \quad \Sigma = \{+, \times, (, ), 1, 2, 3\}
\]
\[
E \rightarrow E \times E
\]
\[
E \rightarrow 1 \mid 2 \mid 3
\]

\( L(G_{\text{Arith}}) \) has arithmetic expressions with parentheses, e.g.
\[
(1 + 2) \times 3
\]
\[
E \rightarrow E \times E \rightarrow (E + E) \times E \rightarrow (1 + E) \times E
\]
\[
\rightarrow (1 + 2) \times E \rightarrow (1 + 2) \times 3
\]
Example

\[ G_{\text{Arth-Int}}: \]
\[ E \rightarrow (E+E) \]
\[ E \rightarrow EXE \]
\[ E \rightarrow I \]
\[ I \rightarrow 0 | 1T | 2T | \ldots | 9T \]
\[ T \rightarrow \& | T0 | T1 | \ldots | T9 \]

This generates expressions with decimal integers
e.g. \( (214 + 109) \times (51080 + 76541) \)

It is easy to show that every regular language is context free.

**Theorem** Every regular language is context free.

**Proof** (by example): Suppose a language \( L \) is recognized by a DFA such as

![DFA Diagram]
The following grammar $G$ "simulates" the DFA.

$$
\begin{align*}
G: & \quad Q_1 \rightarrow 0 Q_2 \mid 1 Q_1 \\
& \quad Q_2 \rightarrow 0 Q_5 \mid 1 Q_3 \\
& \quad Q_3 \rightarrow 0 Q_3 \mid 1 Q_6 \\
& \quad Q_4 \rightarrow 0 Q_1 \mid 1 Q_2 \\
& \quad Q_5 \rightarrow 0 Q_5 \mid 1 Q_4 \\
& \quad Q_6 \rightarrow 0 Q_5 \mid 1 Q_6
\end{align*}
$$

\begin{align*}
Q_5 & \rightarrow \varepsilon \\
Q_6 & \rightarrow \varepsilon
\end{align*}

Simulate transitions of the DFA

Add these only for accept states of the DFA.

Here $Q_1, Q_2, \ldots, Q_6$ are variables of the grammar and $Q_1$ is the start variable (state).

Exercise convince yourself that $L = L(G)$!

It is easily seen that the class of CFLs is closed under $\cup, \cdot, \mathsf{K}$.

Theorem If $L_1, L_2$ are context-free languages, then so are $L_1 \cup L_2, L_1 \cdot L_2, L_1^*$. 
Proof: Let the grammars for \( L_1, L_2 \) be \( G_1, G_2 \) respectively.

\[
G_1: S \rightarrow S_1 \\
G_2: S \rightarrow S_2
\]

Where \( S_1, S_2 \) are their start variables respectively. We may assume that their sets of variables are disjoint (but the set of terminals \( \Sigma \) is the same and \( L_1, L_2 \) are languages over \( \Sigma \)). The grammar \( G_1 \) with a new start variable \( S \), for languages \( L_1 \cup L_2, L_1 \circ L_2, L_1^* \) (as the case may be) is:

\[
L_1 \cup L_2 \quad L_1 \circ L_2 \quad L_1^*
\]

\[
S \rightarrow S_1 | S_2 \\
G_1: \quad S_1 \rightarrow \cdot \\
G_2: \quad S_2 \rightarrow \cdot \\
\]

\[
S_1 \rightarrow \cdot \\
G_1: \quad S \rightarrow S_1 S_2 \\
G_2: \quad S_2 \rightarrow \cdot \\
\]

\[
S_1 \rightarrow \\
G_1: \quad S \rightarrow S_1 S | \epsilon \\
\]
Ambiguity of Grammars

Consider the grammar:

\[ G_{\text{Add-Mult}} : \]

\[ E \rightarrow E + E \]
\[ E \rightarrow E \times E \]
\[ E \rightarrow 11213 \]

and the string \( 1 + 2 \times 3 \) generated by it. There are two distinct parse trees that yield this string.

![Parse Trees](image)

It is not difficult to see that every parse tree corresponds to a unique "leftmost derivation" and vice versa. Here:

**Def.** A derivation is said to be a leftmost derivation if at every step, the leftmost variable is substituted for using a grammar rule.
The leftmost derivations corresponding to the parse trees above are, respectively:

\[
\begin{align*}
E & \quad \Downarrow \\
E + E & \quad \Downarrow \\
1 + E & \quad \Downarrow \\
1 + E \times E & \quad \Downarrow \\
1 + 2 \times E & \quad \Downarrow \\
1 + 2 \times 3 & \quad \Downarrow
\end{align*}
\]

\[
\begin{align*}
E & \quad \Downarrow \\
E \times E & \quad \Downarrow \\
E + E \times E & \quad \Downarrow \\
1 + E \times E & \quad \Downarrow \\
1 + 2 \times E & \quad \Downarrow \\
1 + 2 \times 3 & \quad \Downarrow
\end{align*}
\]

The two derivations are "intrinsically different." If semantics are attached to the terminals +, \times then these amount to whether in the expression 1+2\times3, the addition is evaluated first or the multiplication.

The grammar is said to be ambiguous.

\textit{Def.} A grammar \( G \) is ambiguous if there is some string \( w \in L(G) \) that has two (or more) different leftmost derivations (or equivalently parse trees).
It is possible to rewrite this grammar as below so that the new grammar is unambiguous.

\[ G'_{\text{Add mult}} : \begin{align*}
E & \rightarrow S \\
S & \rightarrow S + P / P \\
P & \rightarrow P \times D \mid D \\
D & \rightarrow 1 \mid 2 \mid 3
\end{align*} \]

'S' = 'Sum'
'P' = 'Product'
'D' = 'Digit'

The interpretation is that the overall arithmetic expression is viewed as "sum of products". Now the string 1+2\times3 has only one parse tree and a leftmost derivation:

I.e. the parse tree on the left (before) is "preferred".
Exercise Convince yourself that
- \( L(G'_{\text{Add-mul}}) = L(G) \) and
- \( G'_{\text{Add-mul}} \) is unambiguous.

Note We emphasize that \( G'_{\text{Add-mul}} \) may yield two different derivations for the same string, e.g., for the string 1+2

\[
\begin{align*}
E & \to S \\
S & \to S + P \\
P & \to P + P \\
P & \to D + P \\
P & \to D + 2 \to 1 + 2
\end{align*}
\]

\[
\begin{align*}
E & \to S \\
S & \to S + P \\
P & \to S + D \\
P & \to S + 2 \to P + 2 \to D + 2 \to 1 + 2
\end{align*}
\]

However, there is only one leftmost derivation and a parse tree, i.e.,

\[
\begin{align*}
E & \to S \\
S & \to S + P \\
P & \to P + P \\
P & \to D + P \\
P & \to 1 + P \\
P & \to 1 + D \\
P & \to 1 + 2
\end{align*}
\]

This is why we insist on leftmost derivations (or parse trees).
Example  This grammar is ambiguous.

\[ G_{\text{Add}} : \quad E \rightarrow E + E \]
\[ E \rightarrow 1 \mid 2 \mid 3. \]

Since \(1 + 2 + 3\) has two parse trees:

It is easy to rewrite an equivalent, unambiguous grammar.

\[ G'_{\text{Add}} : \quad E \rightarrow E + D \mid D \]
\[ D \rightarrow 1 \mid 2 \mid 3 \]

Example  If-then-else grammar as below is ambiguous.

\[ S \rightarrow \text{if } E \text{ then } S \]
\[ S \rightarrow \text{if } E \text{ then } S \text{ else } S \]
\[ S \rightarrow \text{other} \]

Here if, then, else, other are terminals.
'S' stands for statement, 'other' stands for other statement, 'E' stands for expression (or
logical condition) that we don't really care about. Consider the following statement:

if $E_1$ then if $E_2$ then other, else other_2

There are two ways to (semantically) interpret the statement:

if $E_1$ then (if $E_2$ then other_1, else other_2)

if $E_1$ then (if $E_2$ then other_1) else other_2

These lead to two different parse trees:

Hence the grammar is ambiguous.
There is a text-book manner to rewrite an equivalent unambiguous grammar.

\[ S \rightarrow \text{Matched-}S \mid \text{Unmatched-}S \]

\[ \text{Matched-}S \rightarrow \text{if } E \text{ then } \text{Matched-}S \text{ else } \text{Matched-}S \]

\[ \text{Unmatched-}S \rightarrow \text{if } E \text{ then } S \]

\[ \text{if } E \text{ then } \text{Matched-}S \text{ else } \text{Unmatched-}S \]

The statement before has the parse tree:

```
  S
  /\|
/   \  \
|     |
Unmatched-S

/   \
|   |
if E then S

/   \
|   |
Matched-S

/   \
|   |
if E then Matched-S else Matched-S

/   \
|   |
other,

/   \
|   |
other
```

I.e. the left parse tree (before) is "preferred".
Sometimes it is impossible to write a grammar for a CFL that is unambiguous.

\[ \text{Def} \quad \text{A CFL is called inherently ambiguous if every grammar for it is ambiguous.} \]

Example \[ L = \{ a^i b^j c^k \mid i=j \text{ or } j=k, \ i,j,k \geq 0 \} \]

is inherently ambiguous. We'll not prove this in this class.

Note \[ L_1 = \{ a^i b^j c^k \mid i=j \} \]

\[ L_2 = \{ a^i b^j c^k \mid j=k \} \] are both context-free.

However, as we prove later,

\[ L_1 \cap L_2 = \{ a^i b^j c^k \mid i=j=k \} \]

is not context-free. Thus the class of CFLs is not closed under intersection and hence also not closed under complements (why? \( \overline{A \cup B} = \overline{A} \cap \overline{B} \)). In this regard, CFLs differ from regular languages.