Ambiguity in Grammars and Languages

In the grammar

1. \( E \rightarrow I \)
2. \( E \rightarrow E + E \)
3. \( E \rightarrow E * E \)
4. \( E \rightarrow (E) \)

... 

the sentential form \( E + E * E \) has two derivations:

\[
E \Rightarrow E + E \Rightarrow E + E * E
\]

and

\[
E \Rightarrow E * E \Rightarrow E + E * E
\]

This gives us two parse trees:

(a)  
\[
\begin{array}{c}
E \\
E + E \\
E * E \\
E
\end{array}
\]

(b)  
\[
\begin{array}{c}
E \\
E * E \\
E + E \\
E
\end{array}
\]
The mere existence of several *derivations* is not dangerous, it is the existence of several parse trees that ruins a grammar.

Example: In the same grammar

5. $I \rightarrow a$
6. $I \rightarrow b$
7. $I \rightarrow Ia$
8. $I \rightarrow Ib$
9. $I \rightarrow I0$
10. $I \rightarrow I1$

the string $a + b$ has several derivations, e.g.

\[ E \Rightarrow E + E \Rightarrow I + E \Rightarrow a + E \Rightarrow a + I \Rightarrow a + b \]

and

\[ E \Rightarrow E + E \Rightarrow E + I \Rightarrow I + I \Rightarrow I + b \Rightarrow a + b \]

However, their parse trees are the same, and the structure of $a + b$ is unambiguous.
**Definition:** Let $G = (V, T, P, S)$ be a CFG. We say that $G$ is *ambiguous* if there is a string in $T^*$ that has more than one parse tree.

If every string in $L(G)$ has at most one parse tree, $G$ is said to be *unambiguous*.

Example: The terminal string $a + a * a$ has two parse trees:

(a) 
\[
E \quad \begin{array}{c}
E + E \\
I E * E \\
E I I \\
a I a
\end{array}
\]

(b) 
\[
E \quad \begin{array}{c}
E * E \\
E + E I \\
I I a \\
a I a
\end{array}
\]
Removing Ambiguity From Grammars

Good news: Sometimes we can remove ambiguity “by hand”

Bad news: There is no algorithm to do it

More bad news: Some CFL’s have only ambiguous CFG’s

We are studying the grammar

\[ E \rightarrow I \mid E + E \mid E \times E \mid (E) \]
\[ I \rightarrow a \mid b \mid Ia \mid Ib \mid I0 \mid I1 \]

There are two problems:

1. There is no precedence between * and +

2. There is no grouping of sequences of operators, e.g. is \( E + E + E \) meant to be \( E + (E + E) \) or \((E + E) + E\).
Solution: We introduce more variables, each representing expressions of same "binding strength."

1. A *factor* is an expression that cannot be broken apart by an adjacent * or +. Our factors are

   (a) Identifiers

   (b) A parenthesized expression.

2. A *term* is an expression that cannot be broken by +. For instance $a \times b$ can be broken by $a1\times$ or $\times a1$. It cannot be broken by +, since e.g. $a1 + a \times b$ is (by precedence rules) same as $a1 + (a \times b)$, and $a \times b + a1$ is same as $(a \times b) + a1$.

3. The rest are *expressions*, i.e. they can be broken apart with * or +.
We’ll let \( F \) stand for factors, \( T \) for terms, and \( E \) for expressions. Consider the following grammar:

1. \( I \to a \mid b \mid Ia \mid Ib \mid I0 \mid I1 \)
2. \( F \to I \mid (E) \)
3. \( T \to F \mid T \ast F \)
4. \( E \to T \mid E + T \)

Now the only parse tree for \( a + a \ast a \) will be

\[
\begin{array}{c}
E \\
\bigg| \bigg| \\
E + T \\
\bigg| \bigg| \\
T T * F \\
\bigg| \bigg| \\
F F I \\
\bigg| \bigg| \\
I I a \\
\bigg| \bigg| \\
a a
\end{array}
\]
Why is the new grammar unambiguous?

Intuitive explanation:

- A factor is either an identifier or \((E)\), for some expression \(E\).

- The only parse tree for a sequence

\[ f_1 \ast f_2 \ast \cdots \ast f_{n-1} \ast f_n \]

of factors is the one that gives \(f_1 \ast f_2 \ast \cdots \ast f_{n-1}\) as a term and \(f_n\) as a factor, as in the parse tree on the next slide.

- An expression is a sequence

\[ t_1 + t_2 + \cdots + t_{n-1} + t_n \]

of terms \(t_i\). It can only be parsed with \(t_1 + t_2 + \cdots + t_{n-1}\) as an expression and \(t_n\) as a term.
Leftmost derivations and Ambiguity

The two parse trees for \( a + a \ast a \)

![Two parse trees](image)

(a) \hspace{1cm} (b)

give rise to two derivations:

\[
E \Rightarrow E + E \Rightarrow I + E \Rightarrow a + E \Rightarrow a + E \ast E \\
\Rightarrow a + I \ast E \Rightarrow a + a \ast E \Rightarrow a + a \ast I \Rightarrow a + a \ast a
\]

and

\[
E \Rightarrow E \ast E \Rightarrow E + E \ast E \Rightarrow I + E \ast E \Rightarrow a + E \ast E \\
\Rightarrow a + I \ast E \Rightarrow a + a \ast E \Rightarrow a + a \ast I \Rightarrow a + a \ast a
\]
In General:

- One parse tree, but many derivations
- Many *leftmost* derivation implies many parse trees.
- Many *rightmost* derivation implies many parse trees.

**Theorem 5.29:** For any CFG $G$, a terminal string $w$ has two distinct parse trees if and only if $w$ has two distinct leftmost derivations from the start symbol.
Sketch of Proof: (Only If.) If the two parse trees differ, they have a node a which different productions, say $A \to X_1X_2\cdots X_k$ and $B \to Y_1Y_2\cdots Y_m$. The corresponding leftmost derivations will use derivations based on these two different productions and will thus be distinct.

(If.) Let’s look at how we construct a parse tree from a leftmost derivation. It should now be clear that two distinct derivations gives rise to two different parse trees.
A CFL $L$ is *inherently ambiguous* if all grammars for $L$ are ambiguous.

Example: Consider $L =$

$$\{a^n b^n c^m d^m : n \geq 1, m \geq 1\} \cup \{a^n b^m c^m d^n : n \geq 1, m \geq 1\}.$$

A grammar for $L$ is

$$S \rightarrow AB \mid C$$
$$A \rightarrow aAb \mid ab$$
$$B \rightarrow cBd \mid cd$$
$$C \rightarrow aCd \mid aDd$$
$$D \rightarrow bDc \mid bc$$
Let’s look at parsing the string $aabbccdd$. 

(a) 

(b)
From this we see that there are two leftmost derivations:

\[ S \Rightarrow AB \Rightarrow aAbB \Rightarrow aabbB \Rightarrow aabbcBd \Rightarrow aabbcdd \]

and

\[ S \Rightarrow C \Rightarrow aCd \Rightarrow aaDdd \Rightarrow aabDcdd \Rightarrow aabbcddd \]

It can be shown that every grammar for \( L \) behaves like the one above. The language \( L \) is inherently ambiguous.