Derivation and Ambiguity

Course Name: Compiler Design
Course Code: CSE331
Level:3, Term:3
Department of Computer Science and Engineering
Daffodil International University
A derivation is basically a sequence of production rules, in order to get the input string. During parsing, we take two decisions for some sentential form of input:

- Deciding the non-terminal which is to be replaced.
- Deciding the production rule, by which, the non-terminal will be replaced.
- To decide which non-terminal to be replaced with production rule, we can have two options.
• **Left-most Derivation**

  If the sentential form of an input is scanned and replaced from left to right, it is called left-most derivation. The sentential form derived by the left-most derivation is called the left-sentential form.

• **Right-most Derivation**

  If we scan and replace the input with production rules, from right to left, it is known as right-most derivation. The sentential form derived from the right-most derivation is called the right-sentential form.
Production rules:

\[ E \rightarrow E + E \]
\[ E \rightarrow E \ast E \]
\[ E \rightarrow \text{id} \]
\[ E \rightarrow \text{id} \]

Input string: \( \text{id} + \text{id} \ast \text{id} \)

The left-most derivation is:

\[ E \rightarrow E \ast E \]
\[ E \rightarrow E + E \ast E \]
\[ E \rightarrow \text{id} + E \ast E \]
\[ E \rightarrow \text{id} + \text{id} \ast E \]
\[ E \rightarrow \text{id} + \text{id} \ast \text{id} \]

The right-most derivation is:

\[ E \rightarrow E + E \]
\[ E \rightarrow E + E \ast E \]
\[ E \rightarrow E + E \ast \text{id} \]
\[ E \rightarrow E + \text{id} \ast \text{id} \]
\[ E \rightarrow \text{id} + \text{id} \ast \text{id} \]
• A parse tree is a graphical depiction of a derivation.
• It is convenient to see how strings are derived from the start symbol.
• The start symbol of the derivation becomes the root of the parse tree.
Constructing the Parse Tree

• Step 1:
  \[ E \rightarrow E \ast E \]

• Step 2:
  \[ E \rightarrow E + E \ast E \]
• Step 3:
  \[ E \rightarrow \text{id} + E \ast E \]

• Step 4:
  \[ E \rightarrow \text{id} + \text{id} \ast E \]
Step 5:
E → id + id * id

• In a parse tree:
  - All leaf nodes are terminals.
  - All interior nodes are non-terminals.
  - In-order traversal gives original input string
• A grammar $G$ is said to be ambiguous if it has more than one parse tree (left or right derivation) for at least one string.

**Example:**

\[
E \rightarrow E + E \\
E \rightarrow E - E \\
E \rightarrow id
\]

For the string $id + id - id$, the above grammar generates two parse trees:
• Parsing that we have seen so far.
Top-Down Parsing

• Parsing is the process of determining if a string of tokens can be generated by a grammar.
• For any context-free grammar there is a parser that takes at most $O(n^3)$ time to parse a string of $n$ tokens.
• **Top-down parsers** build parse trees from the *top (root)* to the *bottom (leaves)*.
• Two top-down parsing are to be discussed:
  o **Recursive Descent Parsing**
  o **Predictive Parsing** An efficient non-backtracking parsing called for LL(1) grammars.
Consider the grammar
\[
S \rightarrow cAd \\
A \rightarrow ab \mid a
\]
Input string \(w = \text{cad}\)

To construct a parse tree for this string using top-down approach, initially create a tree consisting of a single node labeled \(S\).
Example of Top-Down Parsing

S → cAd
A → ab | a
Input string w = cad

Fig 2.5 Steps in top-down parse
Procedure of Top-down Parsing

• An input pointer points to c, the first symbol of w.
• Then use the first production for S to expand the tree and obtain the tree.
• The leftmost leaf, labeled c, matches the first symbol of w.
• Next input pointer to a, the second symbol of w.
• Consider the next leaf, labeled A.
• Expand A using the first alternative for A to obtain the tree.
Now have a match for the second input symbol. Then advance to the next input pointer d, the third input symbol and compare d against the next leaf, labeled b. Since b does not match d, report failure and go back to A to see whether there is another alternative. (Backtracking takes place).

If there is another alternative for A, substitute and compare the input symbol with leaf.

Repeat the step for all the alternatives of A to find a match using backtracking. If match found, then the string is accepted by the grammar. Else report failure.

A left-recursive grammar can cause a recursive-descent parser, even one with backtracking, to go into an infinite loop.

As discussed above, an easy way to implement a recursive descent parsing with backtracking is to create a procedure for each non-terminals.
Transition Diagram for Predictive Parsers

\[ E \rightarrow TE' \]
\[ E' \rightarrow + TE' \mid \varepsilon. \]
\[ T \rightarrow FT' \]
\[ T' \rightarrow * FT' \mid \varepsilon. \]
\[ F \rightarrow (E) \mid \text{id} \]
**Example:** Build the parse tree for the arithmetic expression $4 + 2 \times 3$ using the expression grammar:

\[
E \rightarrow E + T \mid E - T \mid T \\
T \rightarrow T \times F \mid F \\
F \rightarrow a \mid (E)
\]

where $a$ represents an operand of some type, be it a number or variable. The trees are grouped by height.
\[ E \rightarrow E + T | E - T | T \]
\[ T \rightarrow T \times F | F \]
\[ F \rightarrow a | (E) \]
Grammar:

\[ E \to E + E \mid a \]

Derivation:

\[ E \Rightarrow E + \bar{E} \]
\[ \Rightarrow \bar{E} + E + E \]
\[ \Rightarrow a + \bar{E} + E \]
\[ \Rightarrow a + a + \bar{E} \]
\[ \Rightarrow a + a + a \]
Example: Build the parse tree for the arithmetic expression “a+a+a” using the expression grammar:

\[ E \rightarrow E + E \mid a \]
Consider again, the grammar specifying only addition in expression:

\[ E \rightarrow E + E \mid a \]

Left Most Derivation (LMD):

\[
\begin{align*}
E & \Rightarrow \bar{E} + E \\
 & \Rightarrow \bar{E} + E + E \\
 & \Rightarrow a + \bar{E} + E \\
 & \Rightarrow a + a + E \\
 & \Rightarrow a + a + a
\end{align*}
\]

Right Most Derivation (LMD):

\[
\begin{align*}
E & \Rightarrow E + \bar{E} \\
 & \Rightarrow E + E + \bar{E} \\
 & \Rightarrow E + \bar{E} + a \\
 & \Rightarrow \bar{E} + a + a \\
 & \Rightarrow a + a + a
\end{align*}
\]
The grammar is:  \( S \to S \ S \mid (S) \mid \varepsilon \)

Consider the string "(()())"

\[
\begin{align*}
S & \Rightarrow \hat{S} \ S \\
& \Rightarrow (\hat{S}) \ S \\
& \Rightarrow (\hat{S}) \\
& \Rightarrow (\hat{S}) \\
& \Rightarrow (\hat{S}) \\
& \Rightarrow (\hat{S})
\end{align*}
\]
The grammar is:  \( S \rightarrow S\ S \mid (S) \mid \varepsilon \)
Consider the string "((()))"

\[
S \Rightarrow \overset{\varepsilon}{S} \\
\Rightarrow (\overset{\varepsilon}{S})S \\
\Rightarrow ((\overset{\varepsilon}{S}))S \\
\Rightarrow ((()))S \\
\Rightarrow ((()))\overset{\varepsilon}{S} \\
\Rightarrow ((()))((())) \\
\Rightarrow ((()))((()))
\]

AMBIGUOUS
THANK YOU