From HW1, we’ve learned that using different sets of productions, we can get different parse trees for the same expression $3 + 4 \times 5$.

I. $Expr \rightarrow Expr + Term \mid Expr \times Term \mid Term$
   $Term \rightarrow 0 \mid \ldots \mid 9$

```
   Expr
   /|\  \
   / | \  \
  Expr * Term
  / |  \
 Expr + Term 5
 |  |
 Term 4
 |
 3
```

II. $Expr \rightarrow Term + Expr \mid Term \times Expr \mid Term$
   $Term \rightarrow 0 \mid \ldots \mid 9$

```
   Expr
   /|\  \
   / | \  \
 Term + Expr
  / |  \
 3   Term * Expr
   |  \
    |  \
   4   Term
    |  
    |  
    5
```

III. $Expr \rightarrow Expr + Term \mid Term$
   $Term \rightarrow Term \times Factor \mid Factor$
   $Factor \rightarrow 0 \mid \ldots \mid 9$

```
   Expr
   /|\  \
   / | \  \
 Expr + Term
  / |  \
 Expr / \  \
 / | \  \
 Term + Term * Factor
  |  \
   |  \
   |  \
 Factor Factor 5
   |
   |
   3
   4
```

Computers interpret the tree using depth-first traverse. So, these three parse trees are read as

I. $(3 + 4) \times 5$
II. $3 + (4 \times 5)$
III. $3 + (4 \times 5)$
- You can tell that II and III are the correct way to interpret the given expression $3 + 4 \times 5$, but I is not.
- The reason is that the production set of II and III define the precedency of operations + and $\times$ correctly, but the production set of I doesn’t.
- To specify hierarchies of operations, grammars can become extremely complex. C lite uses 13 production rules to specify the hierarchies of boolean operations (||, &&, ==, !=, >, <, >=, <=) arithmetic operations (+, -, x, /, %), and unary operations (-, !).

Flexible & Ambiguous Grammars

- Let’s work on one more example.
- **Example**: Build a parse tree for $3 - 1 + 2$ based on the following rules

\[
\begin{align*}
Expr & \rightarrow Expr \ Op \ Expr | (Expr) | Integer \\
Op & \rightarrow + | - | \star | / \\
Integer & \rightarrow Digit {Digit} \\
Digit & \rightarrow 0 \cdots | 9
\end{align*}
\]

\[
\begin{array}{c}
\text{Expr} \\
\text{Expr} \ Op \ Expr \ + \ 2 \\
\text{Expr} \ Op \ Expr \ - \ Expr \ Op \ Expr
\end{array}
\]

\[
\begin{array}{c}
3 - 1 + 2 \\
(3 - 1) + 2 \\
3 - (1 + 2)
\end{array}
\]

- We can generate two distinct parse tree using above rules.
- We call a grammar **ambiguous grammar** if its language contains at least one string that has two or more distinct parse tree.
- Ambiguous grammars are not good as they may confuse the compilers. Compilers can generate two different results for the same expression.
- However, sometimes, we may want to use an ambiguous grammar to simplify the number of rules requires. In this case, ambiguities in grammars are generally resolved using additional rules.
  - For example, if we have a table of precedence and a default left-to-right ordering of operators of equal precedence, then we can resolve any ambiguities that arise.

- **Dangling else** is a common ambiguity in language syntax.
• When an if statement is contained inside an if statement, which if statement does a subsequent else belong to?

• Consider the following code snippet

```c
if (x < 0)
  if (y < 0)
    y = y + 1;
else
  y = 0;
```

- Without inserting curly braces, it seems like that the else branch could match either if condition. Different languages address this ambiguity differently.

- Solution of C

  • C addresses by including a description in the nature language in its documentation that an else clause is associated with the textually nearest if statement in any ambiguous case.
  • So, the output of the follow code is “there.”

```c
#include <stdio.h>

int main (int args, char *argv[]) {
  int a = -1;
  int b = 1;
  if (a < 0)
    if (b < 0)
      printf("here\n");
  else
    printf("there\n");
}
```

- Solution of Java

  • Clearly defined in grammar to address the ambiguity.
  • It is not permitted that an if statement without an else clause as the single statement after an if.
  • The following code snippet, for example, will not do what the tabbing implies. The actual output is “there”, since Java consider the else branch belongs to the second if statement.

```java
public class Ambiguity {
  public static void main (String args[]) {
    int a = -1;
    int b = 1;
    if (a < 0)
      if (b < 0)
        System.out.println("here");
    else
      System.out.println("there");
  }
}
```
- Solution of Python
  - Require nested if statements to be indented. The actual output is "".

```python
a = -1
b = 1

if (a < 0):
    if (b < 0):
        print "here\n"
    else:
        print "there\n"
```