Concepts of Programming Languages, CSCI 305, Fall 2021
Precedence & Associatively in Context-Free Grammars
Introduction to Parsing, Oct. 4

Precedence & Associatively in Context-Free Grammars

1. Recall the sample grammar:

   assign → id = expr
   id → A | B | C | D
   expr → expr + expr | expr * expr | (expr) | id

   This grammar is ambiguous. Modify the grammar so that it is unambiguous and uses the usual rules. That is, * has higher precedence than +, and both + and * are left associative.

   To make * have higher precedence than +:

   assign → id = expr
   id → A | B | C | D
   ~~expr → expr + expr | expr * expr | (expr) | id~~
   expr → expr + expr | term
   term → term * term | factor
   factor → (expr) | id

   To make left associativity:

   assign → id = expr
   id → A | B | C | D
   expr → expr + term | term
   term → term * factor | factor
   factor → (expr) | id

2. Problem 2.19, page 107, from the text:
Develop a grammar for the language consisting of all well-formed regular expressions. Arrange for all operators to be left-associative. Give Kleene closure the highest precedence and alternation the lowest precedence.

Definition of well-formed regular expressions:

Basis:

a for any letter a (upper or lower case)

ε (the empty string)

Induction, given that R_1 and R_2 are regular expression:

R_1 ° R_2 (concatenation)

R_1 | R_2 (alternation)

R_1* (Kleene closure)

Use parenthesis to avoid ambiguity

Use your grammar to create a parse tree for the well-formed regular expression:

(a * | a ° a ° b) ° b

Sample answer:

Since | is both in the alphabet of the language and used to describe the context-free grammar, I'll use **'or'** to describe the context-free grammar.

Since alternation has the lowest precedence, I want it to be highest in the tree (so that it is evaluated last). Therefore, I'll start with alternation, then do concatenation, with Kleene closure last.

Grammar:

R → R | S **'or'** S

S → S ° T **'or'** T

T → T* **'or'** a **'or'** b **'or'** ε **'or'** (R)