**Instructions:**

Consider the deterministic CFG below for arithmetic expressions. This CFG implements operator precedence by introducing the T and F non-terminals and by placing the rules with low precedence operations (+ and -) before rules with high precedence operations (\* and /).

S -> S + T | S - T | T

T -> T \* F | T / F | F

F -> N | (S)

N -> 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9

This grammar can be converted into a right recursive CFG by introducing two non-terminals A and B as follows. Notice that all of the non-terminals with multiple rules now have a terminal as the first symbol so we can “look ahead” one symbol to decide which rule to apply.

S -> TA

T -> FB

A -> +TA | -TA | e

F -> N | (S)

B -> \*FB | /FB | e

N -> 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9

**Questions:**

1. [15 points] Using the CFGs above as examples, create a right recursive CFG for Boolean expressions. Type your CFG into JFLAP and test it with the same Boolean expressions you used in the previous question.

**Answer:**

S 🡪 TA

T 🡪 FB

A 🡪 &TA | |TA | e

F 🡪 N | V

B 🡪 >F | <F | =F

N 🡪 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 |

V 🡪 a | b | c | d

A screenshot of a computer

Description automatically generated

**Challenge Question: [required for graduates, bonus for undergraduates]**

1. Implement a recursive descent parser for Boolean expressions based on your right recursive CFG from the previous question. Your recursive functions should return ‘true’ or ‘false’ to indicate that the syntax of the Boolean expression is correct or incorrect. You do not need to evaluate the Boolean expression. Test your program with the same Boolean expressions you used for the previous question.

A screenshot of a computer

Description automatically generated