CAPSTONE2.0 – Read, Write, now Arithmetic

Congratulations! You've successfully made your language a little more *muchier* by implementing the following grammars:

Plain text for EBNF symbols | **Green for individual grammars** | **Bold Red for Tokens**

program → **PROGRAM** **IDENT** **SEMICOLON** block

block → [**VAR** **IDENT** **COLON** (**REAL** | **INTEGER**) **SEMICOLON**
{ **IDENT** **COLON** (**REAL** | **INTEGER**) **SEMICOLON** }]
compound

Or if you prefer to look at it as a one liner:

[**VAR** **IDENT** **COLON** (**REAL** | **INTEGER**) **SEMICOLON** { **IDENT** **COLON** (**REAL** | **INTEGER**) **SEMICOLON** }] compound

Statements

statement → assign | compound | read | write

assign → **IDENT** **ASSIGN** value

compound → **TOK_BEGIN** statement { **SEMICOLON** statement } **END**

read → **READ** **OPENPAREN** **IDENT** **CLOSEPAREN**

write → **WRITE** **OPENPAREN** (**IDENT** | **STRINGLIT**) **CLOSEPAREN**

Value???

value → **FLOATLIT** | **INTLIT** | **IDENT**

These gave your language the ability to **declare variables**, **read input**, and **write output**. That's a solid foundation! But now we want to level up again. Let's add the ability to do simple arithmetic operations like add, subtract, increment, decrement, multiply, divide, modulo, and exponentiation.

Well to do this we will need to **reconsider what is a "value"?**

Where We're Going

Before now value could be any single "number" essentially. But if we are adding math into the mix, then wouldn't or shouldn't we consider value might be more?

Take the following program for example:

```
PROGRAM MATH;
VAR  A : INTEGER;  B : INTEGER;  C : REAL;
BEGIN
 A := 10;
 B := 3;
 C := (A + B ^^ 2) / 5.0;
 WRITE('Results:');
 WRITE(A + B);
 WRITE(C)
END
```

We are good until we get to variable C. Now we must consider what if math happens?

Consider the following new grammars:

value → term { (**PLUS** | **MINUS**) term }

term → factor { (**MULTIPLY** | **DIVIDE** | **MOD** | **CUSTOM_OPER**) factor }

factor → [**INCREMENT** | **DECREMENT** | **MINUS**] primary

primary → **FLOATLIT** | **INTLIT** | **IDENT** | (**OPENPAREN** value **CLOSEPAREN**)

With these new grammars you can math all the classics and still keep precedence for order of operations.

The Same Song and Dance

1. Start with your rules.l and lexer.h. You will need to add to lexer.h for the first time to be able to twerk the language into your class-voted design
    a. For lexer.h:
        i. Add defines for INCREMENT and DECREMENT
        ii. Add cases for the switch for INCREMENT and DECREMENT
    b. For rules.l:
        i. Add rules for arithmetic operators: +, -, *, /, MOD
        ii. Add rules for CUSTOM_OPER, INCREMENT, and DECREMENT
2. Move on to ast.h and decide on the shape the structures will need to take to be able to print and interpret all the new grammars correctly. Consider these:
    a. Member variables: what at a minimum does this node need to be understood correctly?
    b. How might each node be printed to be understood in our parse tree?
    c. How would one interpret this node correctly?
3. Now the parser, the last piece of the process. Let's look at our grammars again and make a parser for each grammar as we see fit.
    a. Make a node.
    b. What do I expect to see first?
        i. Only one possible thing
            1. Expect that thing
        ii. Several possible things
            1. Use if(peek()==whatIsToCome) expect whatIsToCome, else….
    c. Do I need to store something in a member variable or temp? If so, do it.
    d. What comes next? Rinse and repeat the process.
    e. Done? Return the node when finished.

Demos

There are several test cases available in Canvas for download. Some are built to succeed with our grammars and some are built to PURPOSEFULLY fail when our grammars are interpreted correctly.

We will ask you to demonstrate some of the successful tests for us and ask you to explain why the other failures may fail. A successful demo is one where we feel you have learned the goals and accomplished the objectives to our satisfaction. Perfection is not required but demonstratable understanding is.

Part 3 Report Prompts *(5 prompts × 20 pts = 100 pts total)*

---

1. Redefining "Value" (20 pts)

Explain how your definition of **value** changed from Part 2 to Part 3.
Why did we replace it with grammars like term, factor, and primary, and what new possibilities does that open for your language?

2. Order of Operations & Precedence (20 pts)

Describe how your grammar enforces **operator precedence** and **associativity**.
How do your parsing rules ensure that multiplication/division/mod happen before addition/subtraction, and how do parentheses alter that order?

3. Unary, Binary, and Ternary Thinking (20 pts)

Define what **unary**, **binary**, and **ternary** operations mean in programming languages. Which of these does your language currently support, and how are they handled differently in parsing or interpretation? If you were to add a ternary operator later, what kinds of grammar or precedence changes would it need?

4. What Comes Next — Beyond Arithmetic (20 pts)

Now that your language can perform arithmetic, what new kinds of operations should it learn next? Discuss how **relational**, **truth**, or **equality** operators (like <, >, =, <>, AND, OR, NOT) could expand what your programs can express. How might these require changes to your grammar or AST design?

5. Reflection — Your Math Journey (20 pts)

Which part of implementing arithmetic challenged you most — tokenizing operators, writing recursive grammar loops, or interpreting results? How did you debug or reason through those challenges, and what concept finally *clicked* for you by the end of Part 3?