William Hellems-Moody

CS 3210 – 001

Programming Project 04 [Last]

Due December 10, 2023

Final Programming Project 04

Documentation: PDF README

Parser's Usage:

The WHMLexer parser program created for program 04 was designed to handle the evaluation and computation of mathematical expressions. The program vividly supports arithmetic operators `(+, -, *, /)`, implemented within prior programming projects. The program also supports handling expressions containing logical operations `(AND, OR, NOT)`. This means that expressions such as `NOT(0 AND 1) OR (3 == 3)` as True, which the Lexer program can successfully handle by checking to parse the expression and checking if each of the keywords used within the expression is alpha characters within the set of character `[a-Z]`. Finally, the program also contains error-checking for improper expressions entered, along with being able to handle expressions containing both integers and floating-point numbers.

Assumptions:

My parser program assumes that each input is a well-defined and adequately structured expression that adheres to the syntax rules defined by the program. For this program, the program was set up in a way to handle and check for logical operators `(AND, OR NOT)` entered in uppercase, along with relational operators `(==, !=, <, >, <=, >=)`. Finally, this program handles the correct order of operations and precedence within each expression. This is seen when testing expressions such as `3 * (2 + 4) / 2,` where the expression wrapped within the parenthesis takes precedence over the remaining operands and operations as it is evaluated first, ultimately yielding the value 9 once the expression is fully evaluated. Likewise, if the expression is evaluated without parenthesis, the result would yield a value of 8. This is because following PEMDAS, multiplication is evaluated first as `"3 * 2"`, then division as `"4 / 2"` and lastly, `"6 + 2"` yielding the value of `8` as our final result.

William Hellems-Moody

CS 3210 – 001

Programming Project 04 [Last]

Due December 10, 2023

Five Test Cases:

1. **Basic Arithmetic**: "`4 * (7 + 3) / 4`"
   - Expected Result: 10
   - The parser correctly handles the order of operations, first evaluating 7 + 3, followed by multiplying by 4, then dividing by 4 to yield the final result of 10.

2. **Logical Operations**: "`NOT (1 AND 0) OR (3 == 3)`"
   - Expected Result: `True`
   - This test case ensured that the custom parser program is capable of correctly interpreting `AND`, `OR`, and `NOT` logical operations, along with relational equality.
   - From left to right the expressions within parenthesis are evaluated first yielding `NOT (False) OR (True)`. Next, the logical operation performed is `NOT False`, which yields the value of `True`, then finally, evaluating `True OR True` ultimately yields the result of `True` for the entire expression.

3. **Relational Operations**: "`(5 > 3) AND (2 <= 2)`"
   - Expected Result: True
   - This test checks the parser's ability to evaluate greater than, less than, greater than or equal to, and less than or equal to operations.

4. **Division by Zero**: "`10 / (5 - 5)`"
   - Expected Result: Error Message (Division by Zero)
   - This test validates the parser's error handling for division by zero scenarios. For this example, using the parenthesis to give the partial expression "`5-5`" precedence allows us to compute the denominator first, which yields the result `0`. Given that the program has a predefined method in our error class named

“`DivisionByZeroError`” we can handle instances and raise error for expression that meet this criterion.

5. **Unary Operators**: “`4 + -3`”
    - Expected Result: `1`
    - The parser can identify and handle the introduction of unary operators such as `-3`. The program using these values also maintains the correct order of operations as each expression is evaluated to produce the desired result. For this test-case example the expression is evaluated as `4 + (-3)` which equals the result of `1`. This showcases the program’s ability to evaluate expression using these specialized operators.