Assignment: Arithmetic Expression Parser

CS2201: Formal Languages and Automata Theory

March 30, 2026

Objective

Implement a program that parses and evaluates arithmetic expressions consisting of:

- Integer or floating-point numbers (e.g., `3`, `2.5`, `-7`)
- Operators: `+` (addition), `-` (subtraction), `*` (multiplication), `/` (division), `^` (exponentiation)
- Parentheses `(` and `)` for grouping

Your program must first verify that parentheses in the expression are balanced before evaluating it. If parentheses are unbalanced, an appropriate error message should be printed and evaluation should not proceed.

Requirements

1. **Operator Precedence and Associativity:**
   - `^` (exponentiation) has the highest precedence and is **right-associative**.
   - `*`, `/`, and `%` have next precedence and are **left-associative**.
   - `+` and `-` have the lowest precedence and are **left-associative**.
2. **Parentheses Checking:** Before any parsing, your program must check that every opening parenthesis `(` has a matching closing parenthesis `)` and that they are correctly nested. This check must be performed before tokenization or validation of operators/operands. If parentheses are unbalanced, output `Error: Unbalanced parentheses` and exit. Even if the expression contains other errors, the parentheses error must be reported first.
3. **Evaluation:** After verifying balanced parentheses, parse and evaluate the expression according to the precedence and associativity rules. Support numbers that may be integers or decimals. Division should be performed as floating-point division, and then the result should be **truncated toward zero** to an integer. For example, `10/3` yields `3`, `-10/3` yields `-3`. The modulo operator `%` returns the remainder after division, with the same sign as the dividend (left operand) and satisfying: `a = b * (a // b) + (a % b)`, where `//` is integer division truncated toward zero. For example:
   - `10 % 3 = 1`
   - `-10 % 3 = -1` (since -10 = 3*(-3) + (-1))
   - `10 % -3 = 1` (since 10 = (-3)*(-3) + 1)
   - `-10 % -3 = -1` (since -10 = (-3)*3 + (-1))
4. **Unary Minus:** The unary minus operator `-` is allowed at the beginning of an expression and immediately after an opening parenthesis or another operator. For example, `-5+3`, `2*(-3)`, and `2+-3` are valid. However, two consecutive operators without an operand (e.g., `2++3`) are invalid and should trigger `Error: Invalid expression`.

5. **Implicit Multiplication:** When a number or a closing parenthesis is directly followed by an opening parenthesis, it should be interpreted as multiplication. For example:

   - `2(3+4)` → `2*(3+4)`
   - `(2+3)(4+5)` → `(2+3)*(4+5)`
   - `2(3)(4)` → `2*3*4`

   Note: `2(3)` is valid and equals `6`. The same rule applies for numbers: `2.5(3)` is valid.

6. **Error Handling:** Your program should gracefully handle invalid expressions. If multiple errors exist, the following order determines which error message is printed:

   (a) **Unbalanced parentheses** – detected first by the pre-check.

   (b) **Division by zero** – if division or modulo by zero occurs during evaluation.

   (c) **Invalid expression** – for any other syntax error (e.g., two operators in a row, missing operands, invalid characters).

   Use the exact error messages:

   - `Error: Unbalanced parentheses`
   - `Error: Division by zero`
   - `Error: Invalid expression`

7. **Input/Output Format (Important for Automated Testing):**

   - The program must read a single line from **standard input** (stdin). The line may contain the arithmetic expression, possibly with whitespace.
   - It must print the result to **standard output** (stdout).
   - For integer results, print the integer without a decimal point.
   - For division and other operations that produce a non-integer result, output the result truncated toward zero (i.e., discard the fractional part). For example, $10/3$ should print 3, $-10/3$ should print -3.
   - Do not print any extra prompts, messages, or debugging information. Only the result (or an error message) should be printed, followed by a newline.

Running Your Program

Your program will be executed from the command line. It must read a single line from standard input (stdin) and print the result (or an error) to standard output (stdout). For example:

```
$ python 2411AI09_parser.py
3 + 4 * 2
11
```

Suggested Code Structure

A simple Python implementation might look like this:

```
import sys

def evaluate(expr):
    # your parser logic here
    # return result as string (or error message)
```

```
    ...

def main():
    line = sys.stdin.readline()
    if not line:
        return
    expr = line.rstrip('\n')
    result = evaluate(expr)
    print(result)

if __name__ == "__main__":
    main()
```

- **Do not print any extra text** (e.g., "Enter expression:"). The automated grader expects only the output.
- The program must handle whitespace and correctly format numeric results as described above.
- All error messages must match exactly the strings given in the examples.

Examples

Valid Expressions

```
Input:  3 + 4 * 2
Output: 11

Input:  (1 + 2) * 3 ^ 2
Output: 27

Input:  2 ^ 3 ^ 2
Output: 512

Input:  10 / 3
Output: 3

Input:  -10 / 3
Output: -3

Input:  -5 + 3
Output: -2

Input:  2(3+4)
Output: 14

Input:  (2+3)(4+5)
Output: 45

Input:  10 % 3
Output: 1

Input:  -10 % 3
Output: -1

Input:  10 % -3
```

```
Output: 1

Input:  -10 % -3
Output: -1

Input:  2 * (3 + 4) % 5
Output: 4   (since 2*7=14, 14%5=4)
```

Invalid Expressions

```
Input:  (3 + 4 * 2
Output: Error: Unbalanced parentheses

Input:  3 + * 4
Output: Error: Invalid expression

Input:  5 / 0
Output: Error: Division by zero

Input:  5 % 0
Output: Error: Division by zero

Input:  2++3
Output: Error: Invalid expression

Input:  2(3+4
Output: Error: Unbalanced parentheses   (priority: parentheses first)

Input:  (3 ++ 4
Output: Error: Unbalanced parentheses   (parentheses error takes priority)
```

Submission

- **File Name:** Submit a single Python source file named exactly `<rollnumber>_parser.py`. For example, if your roll number is `2411AI09`, the file name must be `2411AI09_parser.py`. Any deviation (including uppercase/lowercase differences, missing underscore, wrong extension) will result in a grade of **0** for the assignment.
- **File Type:** Only `.py` files are accepted. Do not submit compressed archives (ZIP, RAR, etc.), text files, or any other format. Submissions in any other format will receive **0** marks.
- **Code Header:** Include a comment at the top of the file with your full name and roll number.
- **No External Dependencies:** Your code must run without requiring any additional libraries beyond the standard library of the language used.