Reflection – Arithmetic Expression Mini-Language

Author: Alana Bernardez Banegas
Course: Languages & Paradigms – Fall 2025
Instructor: Dr. Omar
Project Category: A2 – BNF Grammar for Mini-Language

Introduction

This project implements a complete arithmetic expression mini-language designed to demonstrate my understanding of key concepts from the Languages & Paradigms course. The assignment required designing a formal BNF grammar, implementing a tokenizer, creating a recursive-descent parser, constructing an Abstract Syntax Tree (AST), and developing an evaluator capable of executing expressions.

This reflection describes the design decisions I made, the structure of each component, the language paradigms demonstrated, the challenges encountered, and the lessons I learned through developing a working interpreter from scratch.

Language Definition Using BNF

The project began with designing a formal BNF grammar to define valid arithmetic expressions. My grammar supports:

- Integer and floating-point numbers
- Parentheses
- Unary operators: + and -
- Binary operators: +, -, *, /, %, ^
- Proper operator precedence
- Right-associative exponentiation

The BNF grammar functions as a precise specification for the language. It is fully independent of implementation and aligns with course concepts relating to formal language theory, syntax specification, and context-free grammars.

Lexer Design (Tokenization)

The lexer reads the input string and converts it into a sequence of tokens. Each token records:

- Its type (e.g., NUMBER, PLUS, CARET)
- The literal lexeme
- A numeric value (for numbers)
- The character position for error reporting

Key design decisions include:

- Supporting both integer and floating-point numbers
- Ignoring whitespace entirely
- Raising descriptive LexerError exceptions for invalid characters
- Tracking positions to produce precise error messages

The lexer represents the first stage of language interpretation and mirrors the lexical analysis techniques used in full-scale languages such as Python, C, and Java.

Parser Design – Recursive Descent

The parser converts the token sequence into an AST. I implemented a recursive-descent parser where each grammar rule corresponds to a specific parsing method:

- _parse_expr() handles addition and subtraction
- _parse_term() handles multiplication, division, and modulo
- _parse_power() handles exponentiation
- _parse_unary() handles unary operators

- _parse_primary() handles numbers and parenthesized expressions

Recursive descent was chosen because it maps naturally onto the BNF grammar, is easier to debug, and provides fine-grained control over precedence and associativity. This parsing strategy reflects concepts studied in class related to compiler front-end design and recursive grammar interpretation.

Abstract Syntax Tree (AST)

The parser builds an AST using three node types:

- **NumberExpr** – stores numeric literals
- **UnaryExpr** – stores unary operators and their operand
- **BinaryExpr** – stores binary operators and left/right subexpressions

The AST captures the hierarchical structure of an expression. For example, the expression (1 + 2) * 3 is represented as a multiplication node whose children are an addition node and the number 3. This design demonstrates object-oriented principles including abstraction and encapsulation.

Evaluation Phase

The evaluator recursively traverses the AST and computes the final value of the expression. Its responsibilities include:

- Evaluating child nodes
- Applying the corresponding operator
- Handling both integer and floating-point arithmetic
- Supporting right-associative exponentiation (e.g., 2^3^2 = 2^(3^2))
- Detecting division-by-zero or modulo-by-zero errors

By separating parsing and evaluation, the project reflects the classic interpreter architecture used in many scripting languages.

Error Handling

Three categories of errors are handled:

- **LexerError** – produced by invalid characters
- **ParseError** – produced by malformed or incomplete syntax
- **EvaluatorError** – produced by runtime issues such as division by zero

All error messages include the position of the failure and a clear descriptive explanation. This structured approach demonstrates the importance of meaningful diagnostics in language tools.

Testing Strategy

The program includes a test suite that automatically validates the correctness of the interpreter.

Valid expression tests include:

- Operator precedence
- Parentheses
- Exponentiation chaining
- Mixed integer/float arithmetic
- Unary operators
- Modulo operations

Invalid expression tests include:

- Missing parentheses
- Incomplete expressions
- Unsupported operators
- Alphabetic characters

Running tests before launching the REPL allowed me to verify correctness and robustness across various scenarios.

Paradigms and Concepts Demonstrated

Formal Language Theory

The BNF grammar demonstrates syntax specification and context-free language design.

Imperative Programming

The lexer, parser, evaluator, and REPL rely on imperative control flow.

Object-Oriented Programming

AST nodes are implemented using classes and encapsulation.

Recursive Programming

Both parsing and evaluation rely heavily on recursion to process nested expressions.

Interpreter Architecture

The project follows the canonical interpreter pipeline:
Source → Lexer → Parser → AST → Evaluator → Output

Error Handling Paradigms

The separation of lexical, syntactic, and semantic errors reflects real-world language design principles.

Challenges and How I Solved Them

Operator Precedence

I encoded precedence directly into the grammar structure by splitting the parser into multiple levels (expr, term, power, unary).

Right-Associative Exponentiation

Exponentiation required special handling so that expressions like 2^3^2 evaluate as 2^(3^2). I solved this using a recursive rule inside the power parser function instead of iterative chaining.

Meaningful Error Reporting

By tracking token positions and including them in exceptions, the interpreter provides precise feedback about syntax failures.

Mixed-Mode Arithmetic

Storing all values internally as floats simplified evaluation while still preserving support for integer inputs.

What I Learned

This project gave me in-depth experience with:

- Designing grammars
- Implementing lexical analyzers
- Building recursive-descent parsers
- Constructing and evaluating ASTs
- Handling multi-stage error detection
- Understanding how expressions are processed internally by compilers

It also deepened my appreciation for how real programming languages implement parsing and interpretation.

Future Enhancements

If extended, I would like to add:

- Variables and assignments
- A symbol table
- Built-in functions such as sin, cos, or max
- Multi-line parsing

- Control structures
- Comment support
- An AST visualization feature

These features would evolve the interpreter into a more complete mini-language.

Conclusion

This project successfully demonstrates core concepts of programming language theory and interpreter design. By defining a grammar, developing a lexer, implementing a recursive-descent parser, constructing an AST, and evaluating expressions, I built a complete interpreter pipeline from scratch. This work strengthened my understanding of parsing, recursion, language semantics, and compiler architecture, making it one of the most insightful and rewarding projects of the course.
