Reflection – Arithmetic Expression Mini-Language

Author: Alana Bernardez Banegas

Languages & Paradigms – Fall 2025

Instructor: Dr. Omar

Project Category: A2 – BNF Grammar for Mini-Language

📌 Introduction

For this project, I implemented a complete arithmetic expression mini-language to demonstrate my understanding of fundamental language and paradigm concepts from the "Languages & Paradigms" course. The assignment required designing a formal BNF grammar, creating a tokenizer, implementing a recursive-descent parser, constructing an AST (Abstract Syntax Tree), and building an evaluator that executes expressions.

This reflection explains how I approached designing the language, why I made certain architectural choices, the paradigms this project demonstrates, challenges I encountered, and what I learned through building a small interpreter from scratch.

📌 Language Definition Using BNF

The first step of this project was designing a formal grammar to define what constitutes a valid arithmetic expression. My BNF grammar supports:

- Integers and floating-point numbers
- Parentheses
- Unary operators (+ and -)
- Binary operators: +, -, *, /, %, ^
- Correct operator precedence

- Right-associative exponentiation

The grammar serves as the “contract” of the language—an exact specification of its syntax. It is completely independent of Python code and represents the structure of the language in a theoretical and implementation-neutral form. This reflects core course concepts regarding formal languages, context-free grammars, and parsing theory.

📌 Lexer Design (Tokenization Phase)

After defining the grammar, I built a lexer that scans raw characters from the input string and produces a list of tokens. Each token contains:

- A type (e.g., `NUMBER`, `PLUS`, `CARET`)
- The original lexeme
- A numerical value (for numbers)
- The character position (for error reporting)

Design decisions:

- Support both integers and floats using digit scanning with optional decimal point.
- Ignore whitespace entirely.
- Raise `LexerError` for unsupported characters.
- Include the expression position in error messages to improve debugging.

The lexer is the first step of interpretation and demonstrates principles of lexical analysis used in real languages such as Python, Java, C++, and JavaScript.

📌 Parser Design – Recursive Descent

To translate the token list into a structured AST, I implemented a recursive-descent parser. Each grammar rule is mapped to one Python function, making the parser readable and directly aligned with the BNF grammar.

Examples:

- `_parse_expr()` handles + and −
- `_parse_term()` handles *, /, %
- `_parse_power()` handles the right-associative ^ operator
- `_parse_unary()` handles unary plus/minus
- `_parse_primary()` handles numbers and parentheses

Recursive descent was chosen because:

1. It directly matches the grammar structure.
2. It is easier to maintain and debug compared to table-driven parsers.
3. It offers full control over associativity and precedence.
4. It perfectly fits the small language required for this project.

This aligns with the course theme of exploring different parsing strategies and understanding how high-level languages parse source code into structured internal representations.

📌 Abstract Syntax Tree (AST)

The parser constructs an AST consisting of three node types:

- NumberExpr – stores numeric values
- UnaryExpr – stores unary operators and their operand
- BinaryExpr – stores binary operators and left/right sub-expressions

The AST represents the hierarchical structure of expressions. For example, `(1 + 2) * 3` becomes a tree where multiplication is the root, with addition and the constant 3 as children.

By implementing the AST as Python classes, this portion demonstrates object-oriented design and dynamic dispatch through class-based structures.

📌 Evaluation Phase (Interpreter)

The evaluator walks the AST recursively:

- Evaluate child nodes
- Apply operators
- Return numeric results

Key behaviors:

- Mixed-mode arithmetic (ints and floats)
- Division-by-zero and modulo-by-zero checks
- Right-associative exponentiation (^)
- Floating-point promotion for consistent results

This evaluation phase reflects the interpreter pattern used in many scripting languages, highlighting how code moves from textual form to executable operations.

📌 Error Handling (Lexing, Parsing, Evaluation)

I implemented three custom error types:

- LexerError – invalid characters
- ParseError – incomplete or malformed syntax
- EvaluatorError – runtime errors like division by zero

Each error includes:

- A clear message
- The position in the input
- The token or operator causing the error

This results in meaningful and user-friendly diagnostics, which is critical for language tools.

📌 Testing Strategy

The test suite includes:

✔️ Valid Expressions

- Operator precedence
- Nested parentheses
- Exponentiation chaining
- Mixed integer/float expressions
- Unary operators
- Modulo behavior

✔️ Invalid Expressions

- Missing closing parenthesis
- Unexpected tokens
- Unsupported operators (`//`)
- Alphabetic characters (`abc`)

Testing ensured correctness and helped validate grammar behavior and parser implementation. Running tests before the REPL also confirmed that all core features work properly before user interaction.

📌 Paradigms & Concepts Demonstrated

1. Formal Language Theory

BNF grammar illustrates syntax specification used in compilers and language processors.

2. Imperative Programming

Lexer, parser, and evaluator functions follow imperative control flow.

3. Object-Oriented Programming

AST nodes are implemented using Python classes and data encapsulation.

4. Recursive Programming

Parsing and evaluation both rely heavily on recursive structure, demonstrating how languages implement nested expressions.

5. Interpreter Architecture

The project follows the classic interpreter pipeline:

6. Error Handling Paradigms

Separation of lexical, syntactic, and semantic (runtime) errors is a key concept in programming language design.

📌 Challenges & How I Solved Them

1. Ensuring Correct Operator Precedence

I had to design grammar rules that naturally reflect precedence rather than manually comparing priority values. Using separate rules (expr, term, power) solved this.

2. Right-Associative Exponentiation

^ needed to evaluate like: This required a recursive rule in <power> rather than a loop.

3. Meaningful Error Messages

I included token positions in every error type to help users understand issues in their expressions.

4. Mixed-mode Arithmetic

I decided to store all numbers internally as floats to simplify evaluation while still accepting integers. This required a recursive rule in <power> rather than a loop.

3. Meaningful Error Messages

I included token positions in every error type to help users understand issues in their expressions.

4. Mixed-mode Arithmetic

I decided to store all numbers internally as floats to simplify evaluation while still accepting integers.

📌 What I Learned

Through building this interpreter from scratch, I gained hands-on experience with:

- Designing grammar-based languages
- Understanding compiler front-end phases
- Recursive parsing logic
- AST representation
- How interpreters evaluate expressions
- How different paradigms influence design choices

I now have a deeper appreciation for how real-world languages like Python, Java, and C handle expressions, parse source code, and report errors.

📌 Future Enhancements

If extended in the future, I would like to add:

- Variables and assignments
- A symbol table
- Built-in functions (sin, max, etc.)
- Multi-line program parsing
- Basic control structures
- Support for comments
- Pretty-printing or visualization of AST nodes

These additions would push the project closer to a full mini-programming language.

📌 Conclusion

This project successfully demonstrates the core concepts of programming language design covered in the course. By defining a grammar, building a lexer, writing a recursive-descent parser, constructing an AST, and evaluating expressions, I implemented an entire interpreter pipeline from scratch. This has strengthened my understanding of parsing theory, compiler design, and programming language paradigms, making it one of the most practical and informative projects I've completed.