Technical Report

Module: Language Design and Implementation (6CC509)

Author: [Loukas Theos / 100667535]

Date: [20/03/2025]

This technical report presents a comprehensive breakdown of developing an interpreter for arithmetic and Boolean expressions in Python. It covers tokenization, parsing via the Shunting Yard algorithm, postfix evaluation, and strict type checking. Each implementation stage is thoroughly analyzed, detailing methodologies, challenges, solutions, and reflective insights.

The primary objective was to create an interpreter to evaluate arithmetic and Boolean expressions, showcasing robust parsing and evaluation capabilities. The interpreter follows a modular design ensuring readability, scalability, and easy maintenance.

Stage 1: Basic Calculator (Arithmetic Expression Evaluation)

Objectives

Develop an interpreter capable of:
- Evaluating arithmetic expressions accurately
- Respecting operator precedence and parentheses
- Handling unary negation effectively

Methodology

- Tokenization: Regular expressions tokenize input expressions into meaningful segments.
- Infix to Postfix (*Shunting Yard Algorithm*): Converts standard mathematical notation to postfix notation.
-Postfix Evaluation: Uses a stack to evaluate postfix expressions efficiently.

Challenges & Solutions

- Unary Negation vs. Subtraction: Context-aware parsing.
- Parentheses Mismatch: Explicit parsing checks.

Stage 2: Boolean Logic and Strict Type Safety

Objectives

Extend interpreter capabilities to:
- Process Boolean logic and expressions
- Implement strict type checking
- Clearly separate numeric and Boolean operations

Methodology

- Tokenization of Boolean Values: Identify Boolean literals and logical operators.
- Strict Type Evaluation: Type-safe operations through explicit checks.

Challenges & Solutions

- Python Boolean/Integer Ambiguity: Explicit type checking.
- Mixed Operator Precedence: Expanded rules for arithmetic and logical mixing.

Examples

Valid:
- true and !false → True
- (5 + 2) == 7 → True

Invalid:
- true + 1 → TypeError
- 5 < false → TypeError

The interpreter implementation enhanced comprehension of critical compiler design aspects, including lexical analysis, parsing, and evaluation methodologies. Addressing Python-specific quirks, like type ambiguity, significantly enriched our understanding and improved the interpreter's robustness.

The developed interpreter effectively evaluates arithmetic and Boolean expressions while enforcing strict type safety.