Translator Architecture - Fourward Programming Language

1. System Overview

- Interpreter-based implementation
- Written in Python
- Modular design (lexer, parser, interpreter, environment)

---

2. Core Components

2.1 Lexer

- Converts source code into tokens
- Handles identifiers, literals, operators, and symbols
- Skips whitespace and comments

2.2 Parser

- Constructs an Abstract Syntax Tree (AST)
- Implements operator precedence and expression grammar
- Detects syntax errors and unexpected tokens

2.3 Interpreter

- Traverses and evaluates the AST
- Handles control flow, arithmetic, and variable scope
- Supports built-in functions like `print` and `input`

---

3. Data Structures

3.1 Abstract Syntax Tree (AST)

- Represents hierarchical program structure
- Used for expression evaluation and control flow
- Built from nested node classes (e.g., `BinaryOp`, `IfStatement`)

3.2 Environment (Symbol Table)

- Stores variable bindings across scopes
- Enables nested block evaluation
- Provides symbol resolution during runtime

---

4. Runtime Environment

4.1 Memory Management

- Python-managed memory model
- Environment class handles scoped variable storage
- No manual memory allocation or garbage collection

4.2 Execution Model

- Sequential execution of AST nodes
- Evaluation of expressions and statements
- Support for runtime errors and basic exception handling

---

5. Implementation Details

5.1 Key Classes

- `Token`: Represents individual lexical units
- `Parser`: Builds the AST from token streams
- `Interpreter`: Evaluates the AST
- `Environment`: Tracks variable scopes and bindings

5.2 Important Methods

- Tokenization using regex patterns
- Recursive descent parsing methods (e.g., `expression()`, `statement()`)
- AST evaluation methods for control flow and expressions

---

6. Error Handling

- Syntax errors detected during parsing (e.g., unexpected token)
- Runtime errors such as undefined variables or invalid operations
- Clear error messages with line and column information
- No error recovery mechanisms implemented yet

---

7. Performance Considerations

- Efficient token and AST processing
- Interpreter runs sequentially; suitable for small-scale programs
- No explicit optimization, but easy to extend

---

8. Extensibility

- Parser and interpreter are modular for easy updates
- Planned support for:
  - User-defined functions
  - Arrays and additional data types

- File I/O and standard library utilities
