AingalLang

Technical Documentation

This documentation provides a comprehensive explanation of every class and method/definition in `interpreter.py` for the AinglLang programming language interpreter. It also details how each connects to the grammar defined in ANTLR4, including how ANTLR is used to tokenize and parse English-like syntax into an executable format.

How ANTLR is Used

AinglLang uses ANTLR4 to define its grammar and generate the parser infrastructure.

➤ **Grammar Files**

- *EnglishLangLexer.g4*: Defines tokens like keywords (Set, If, Display, etc.), identifiers, literals, operators.
- *EnglishLangParser.g4*: Defines the syntactic structure using parser rules (program, statement, expression, functionDeclaration, etc.).

➤ **Generated Components**

- **Lexer (EnglishLangLexer)**: Splits input code into tokens.
- **Parser (EnglishLangParser)**: Creates a parse tree from tokens.
- **Visitor (EnglishLangParserVisitor)**: Base class that we extend in Interpreter to walk and evaluate the parse tree.

➤ **Workflow**

1. Source code is read.
2. Lexer tokenizes input based on *EnglishLangLexer.g4*.
3. Parser builds a parser tree from tokens using *EnglishLangParser.g4*.
4. Interpreter visits the tree nodes using `visit___()` methods to evaluate or execute logic.

Lexical Grammar (EnglishLangLexer.g4)

Category	Token	Symbol/Pattern
Keywords	START_PROGRAM	'Start Program'
	END_PROGRAM	'End Program'
	DEFINE_FUNCTION	'Define Function'
	END_FUNCTION	'End Function'
	RETURN	'Return'
	CALL	'Call'
	SET	'Set'
	TO	'to'
	DISPLAY	'Display'
	IF	'If'
	ELSE_IF	'Else If'
	ELSE	'Else'
	END_IF	'End If'
	FOR	'For'
	FROM	'from'
	IN	'in'
	END_FOR	'End For'
	BREAK	'break'
	WHILE	'While'
	END_WHILE	'End While'
	INVERT_MATRIX	'invert'
	TRANSPOSITION	''T'
	PARENT_SCOPE	'parent'
	DCOLON	'::'
Operators	PLUS	'+'
	MINUS	'-'
	MULTIPLY	'*'
	DIVIDED_BY	'/'
	MODULO	'%'
	EQUALS	'=='
	GREATER_THAN	'>'
	LESS_THAN	'<'
	GREATER_EQUAL	'>='
	LESS_EQUAL	'<='

	ADD_TO	'+='
	SUBTRACT_FROM	'-='
	INCREMENT	'++'
	DECREMENT	'--'
	TIMES	'*='
	DIVIDE_FROM	'/='
	NOT_EQUALS	'!='
	AND	'and'
	OR	'or'
	NOT	'not'
Punctuation	SEMICOLON	';'
	COMMA	','
	COLON	':'
	DOT	'.'
	QUOTE	'"'
	LBRACK	'['
	RBRACK	']'
	LBRACE	'{'
	RBRACE	'}'
	LPAREN	'('
	RPAREN	')'
Literals	NUMBER	[-]? [0-9]+ ('.' [0-9]+)?
	STRING	'"' (~["\r\n])* '"'
Types	TYPE_INT	'int'
	TYPE_STRING	'string'
	TYPE_BOOL	'bool'
	TYPE_FLOAT	'float'
	TYPE_MATRIX	'matrix'
	TYPE_VOID	'void'
	TRUE_VALUE	'true'
	FALSE_VALUE	'false'
Functions	POWER_FUNC	'pow'
	SIN_FUNC	'sin'
	COS_FUNC	'cos'
	TAN_FUNC	'tan'
	CTAN_FUNC	'ctan'
Identifiers	IDENTIFIER	[a-zA-Z_][a-zA-Z_0-9]*

Whitespace	WS	[\t\r\n]+ -> skip
Comments	COMMENT	'//' ~[\r\n]* -> skip

Syntax Grammar (EnglishLangParser.g4)

Grammar Structure

Program Structure

program : START_PROGRAM statement+ END_PROGRAM ;

- Every program must begin with Start Program and end with End Program
- Contains one or more statements

Statements

```
statement:
      variableDeclaration
    | reassignment
    | functionDeclaration
    | functionCall
    | returnStatement
    | displayStatement
    | ifStatement
    | loopStatement
    | forLoop
    | whileLoop
    | blockStatement
    | operation
    ;
```

The language supports all fundamental programming constructs including variables, functions, control flow, and I/O.

Core Language Features

1. Variables

variableDeclaration

```
: SET? IDENTIFIER TO expression typeAnnotation?
;
```

- Variables can be declared with optional Set keyword
- Type annotations are optional (type inference supported)
- Example: Set x to 10 int or simply x to 10

2. Expressions

The language supports several expression types:

Numeric Expressions

```
numExpression : numExpression (PLUS|MINUS) term
    | term;

term: term (MULTIPLY|DIVIDED_BY|MODULO) factor
    | factor;

factor
    : PLUS factor
    | MINUS factor
    | functionCall
    | NUMBER
    | scopedIdentifier
    | IDENTIFIER
    | STRING
    | operation
    | LPAREN numExpression RPAREN
    ;
```

Standard arithmetic operations with proper precedence

Supports unary plus/minus

Boolean Expressions

```
boolExpression
    : boolOrExpression
    ;

boolOrExpression
    : boolAndExpression (OR boolAndExpression)*
    ;

boolAndExpression
    : boolNotExpression (AND boolNotExpression)*
    ;

boolNotExpression
    : NOT boolNotExpression
    | boolPrimary
    ;

boolPrimary
    : numExpression comparisonOp numExpression
    | stringExpression (EQUALS | NOT_EQUALS) stringExpression
    | matrixExpression (EQUALS | NOT_EQUALS) matrixExpression
    | LPAREN boolExpression RPAREN
    | TRUE_VALUE
    | FALSE_VALUE
    | IDENTIFIER
    ;
```

Supports logical AND/OR/NOT operations

Comparison operators: ==, !=, >, <, >=, <=

Matrix Operations

```
matrixExpression: (INVERT_MATRIX)? matrixAtom (TRANSPOSITION)?;
matrixAtom: IDENTIFIER | matrixConstruction;

matrixConstruction: LBRACK row (SEMICOLON row)* RBRACK;
row: value (COMMA value)*;
value: NUMBER | IDENTIFIER;
```

Supports matrix inversion (invert) and transposition ('T)

Matrix construction: [[1, 2]; [3, 4]]

String Operations

```
stringExpression: (STRING | IDENTIFIER) ( PLUS (STRING |
IDENTIFIER))*;
```

- String concatenation with + operator

3. Functions

functionDeclaration

: DEFINE_FUNCTION IDENTIFIER LPAREN parameter? RPAREN blockStatement END_FUNCTION ;

- Functions are declared with Define Function
- Optional typed parameters
- Must end with End Function

Built-in Functions

```
builtInFunctions:POWER_FUNC LPAREN numExpression COMMA numExpression
RPAREN
                  |
              (SIN_FUNC | COS_FUNC | TAN_FUNC | CTAN_FUNC )
               LPAREN numExpression RPAREN;
```

- Math functions: pow, sin, cos, tan, ctan

4. Control Flow

Conditional Statements

```
ifStatement: IF LPAREN boolExpression RPAREN (statement |
blockStatement)
             (ELSE_IF LPAREN boolExpression RPAREN (statement |
blockStatement))*
             (ELSE (statement | blockStatement))?;
```

- Supports If, Else If, and Else clauses
- Both single statements and block statements are allowed

Loops

```
forLoop
    : FOR LPAREN forInit? SEMICOLON
              cond=boolExpression SEMICOLON
              forUpdate RPAREN
      forBody;

whileLoop:
           WHILE LPAREN boolExpression RPAREN
           (LBRACE loopStatements+ RBRACE | statement);
```

- C-style for loops with initialization, condition, and update
- Standard while loops
- Supports break statement

5. I/O Operations

```
displayStatement: DISPLAY expression (',' expression)*;
```

- Output with Display command
- Can display multiple comma-separated expressions

Scoping Rules

```
scopedIdentifier
    : (PARENT_SCOPE DCOLON)+ IDENTIFIER
    ;
```

- Access parent scope variables with parent::variable
- Multiple scope levels supported (e.g., parent::parent::variable)

Type System

```
typeAnnotation: TYPE_STRING | TYPE_INT | TYPE_FLOAT | TYPE_BOOL |
TYPE_MATRIX;
```

- Supported types: int, string, bool, float, matrix
- Type annotations are optional (type inference supported)

Error Recovery

The parser includes several features for robust error handling:

- Optional elements (marked with ?)
- Alternative productions (separated by |)
- Proper grouping with parentheses

Example Valid Constructs

1. Variable declaration:

```
Set counter to 0 int
```

2. Function definition:

```
Define Function square(x int) {
      Return x * x
} End Function
```

3. If statement:

```
If (x > 10) {
      Display "Large number"
} Else {
      Display "Small number"
}
```

4. For loop:

```
For (i to 0; i < 10; i++) {
      Display i
}
```

Compilation Passes

The AingalLang interpreter uses a multi-pass approach to process the source code:

1. Lexical Analysis (Tokenization)

- Handled by ANTLR-generated lexer (AingalLangLexer)
- Converts raw source code into tokens
- Identifies keywords, identifiers, literals, and operators
- Example: Set x to 10 → [SET, IDENTIFIER(x), TO, NUMBER(10)]

2. Syntax Analysis (Parsing)

- Handled by ANTLR-generated parser (AingalLangParser)
- Builds a parse tree from tokens according to grammar rules

- Validates program structure
- Example: Validates that Set x to 10 follows the variableDeclaration rule

3. Semantic Analysis (Interpreter Execution)

- Handled by the Interpreter class
- Performs scope resolution
- Type checking
- Symbol table population
- Actual program execution

4. Runtime Execution

- Dynamic evaluation of expressions
- Function calls with proper scope handling
- Memory management

Module Overview: interpreter.py

➤ **Imports**

```
from antlr4 import *
from EnglishLangLexer import EnglishLangLexer
from EnglishLangParser import EnglishLangParser
from EnglishLangParserVisitor import EnglishLangParserVisitor
import math
```

- Imports the ANTLR4 runtime, lexer, parser, visitor, and math module.

Compilation Passes

The AingalLang interpreter uses a multi-pass approach to process the source code:

1. Lexical Analysis (Tokenization)
    - Handled by ANTLR-generated lexer (AingalLangLexer)
    - Converts raw source code into tokens
    - Identifies keywords, identifiers, literals, and operators
    - Example: Set x to 10 → [SET, IDENTIFIER(x), TO, NUMBER(10)]
2. Syntax Analysis (Parsing)
    - Handled by ANTLR-generated parser (AingalLangParser)
    - Builds a parse tree from tokens according to grammar rules
    - Validates program structure
    - Example: Validates that Set x to 10 follows the variableDeclaration rule
3. Semantic Analysis (Interpreter Execution)
    - Handled by the Interpreter class
    - Performs scope resolution
    - Type checking
    - Symbol table population
    - Actual program execution
4. Runtime Execution
    - Dynamic evaluation of expressions
    - Function calls with proper scope handling
    - Memory management

Symbol Table Implementation

The symbol table is implemented through the Scope class hierarchy, which manages variable storage and resolution:

Scope Class Structure

```
class Scope:
    def __init__(self, parent=None):
        self.variables = {}  # Symbol storage
        self.parent = parent  # Reference to parent scope

    def set_variable(self, name, value):
        self.variables[name] = value

    def has_variable(self, name):
        return name in self.variables

    def get_variable(self, name):
        # Checks current scope then parent chain
        if name in self.variables:
            return self.variables[name]
        elif self.parent:
            return self.parent.get_variable(name)
        else:
            raise Exception(f"Variable '{name}' not found")
```

Symbol Table Features

1. **Hierarchical Scoping**
    - Global scope (created at program start)
    - Function scopes (created during function calls)
    - Block scopes (created for control structures)
2. **Variable Resolution**
    - Current scope first
    - Then parent scopes recursively
    - Supports parent:: syntax for explicit parent access

3. **Type Information Storage**
    - Variables store both value and implicit type
    - Type annotations are enforced during declaration

Symbol Table Operations

Operation	Method	Description
Variable Declare	set_variable()	Adds new symbol to current scope
Variable Lookup	get_variable()	Finds symbol in scope hierarchy
Scope Enter	push_env()	Creates new nested scope
Scope Exit	pop_env()	Returns to parent scope
Existence Check	has_variable()	Checks if symbol exists

Example Symbol Table State

Program:

```
Start Program
    Set x to 10
    Define Function foo() {
        Set y to 20
        Display x, y
    }
End Program

# Symbol Table during foo() execution:
{
    'global': {
        'x': 10,
        'foo': <function>
    },
    'foo_scope': {
        'y': 20,
        'parent': <global_scope>
    }
}
```

Class Definitions

1. BreakStatement

- Marker class used to interrupt loop execution.
- Related Rule: breakStatement

2. FunctionReturn(Exception)

- Custom exception to handle early returns from functions.
- Raised in: visitReturnStatement

3. Scope

Manages variables in a scoped environment.

```
def __init__(self, parent=None)
def set_variable(self, name, value)
def has_variable(self, name)
def get_variable(self, name)
```

- Used by Interpreter to model nested variable scopes.
- Related Rules: All rules involving variable access and block scoping.

Class: Interpreter(EnglishLangParserVisitor)

The core class of the interpreter which walks the parse tree and evaluates the program.

Constructor:

```
def __init__(self)
```

Initializes:

```
global_scope, current_scope, memory, variables, functions,
output_lines, call_stack
```

Scope Management

```
def push_env(self)
def pop_env(self)
def set_var(self, name, value)
def get_var(self, name)
def lookup_variable(self, name)
```

Statement Visitors

visitProgram(ctx): Evaluates the root program.

visitStatement(ctx): Dispatch for a generic statement.

visitBlockStatement(ctx): New scoped environment for a code block.

visitDisplayStatement(ctx): Handles Display keyword. Appends stringified output to output_lines.

visitVariableDeclaration(ctx): Supports type annotation and variable declaration. Grammar: variableDeclaration: 'Set' IDENTIFIER 'to' expression typeAnnotation?;

visitReassignment(ctx): Performs operations like Add 5 to x, Subtract, etc. Grammar: reassignment: ('Add'|'Subtract'|'Multiply'|'Divide') expression 'to' IDENTIFIER;

visitBreakStatement(ctx): Returns BreakStatement() to break loops.

Control Flow

visitIfStatement(ctx): Handles if-else-if-else logic. Grammar: ifStatement

visitLoopIfStatement(ctx): Conditional branching inside loops.

visitWhileLoop(ctx): Standard while-loop.

visitForLoop(ctx): Full for-loop with init, condition, update, body.

visitLoopStatements(ctx): Switchboard for statement types inside loops.

Function Handling

visitFunctionDeclaration(ctx): Registers function with its parameters and body. Grammar: functionDeclaration

visitFunctionCall(ctx): Initiates function call.

callFunction(name, args): Handles scope creation, argument binding, return extraction.

visitReturnStatement(ctx): Raises FunctionReturn exception.

Expression Evaluation

visitExpression(ctx): Main dispatcher for any expression.

Arithmetic:

```
def visitNumExpression(ctx): …
```

```
def visitTerm(ctx): …
def visitUnaryPlus(ctx): …
def visitUnaryMinus(ctx): …
def visitFactorNumber(ctx): …
def visitFactorIdentifier(ctx): …
def visitFactorString(ctx): …
def visitFactorParens(ctx): …
def visitFactorFunctionCall(ctx): …
def visitFactorOperation(ctx): …
def visitFactorscopedIdentifier(ctx): …
```

Logical / Boolean:

```
def visitBoolExpression(ctx)
def visitLogicOr(ctx)
def visitLogicAnd(ctx)
def visitLogicNot(ctx)
def visitLogicPrimaryWrap(ctx)
def visitLogicParen(ctx)
def visitTrueLiteral(ctx)
def visitFalseLiteral(ctx)
def visitLogicIdentifier(ctx)
```

Comparisons:

```
def visitNumComparison(ctx)
def visitStringComparison(ctx)
def visitMatrixComparison(ctx)
```

Matrix Operations

visitMatrixExpression(ctx): Handles matrix inversion or transposition.

visitMatrixAtom(ctx): Returns matrix by identifier or construction.

visitMatrixConstruction(ctx): Parses matrix literals.

visitValue(ctx): Parses individual scalar values for matrices.

Utilities:

```
def transpose_matrix(self, matrix)
```

```
def invert_matrix(self, matrix)
```

Built-in Functions

visitBuiltInFunctions(ctx)

Handles:

- `Power of a to b`
- `Sin, Cos, Tan, Ctan`

Others

visitIdentifier(ctx): Looks up basic variable.

visitScopedIdentifier(ctx): Accesses parent scope.

visitOperation(ctx): Handles increment/decrement (Increment x, Decrement y).

evaluateBinaryOp(left, right, op): Handles all math and comparison operators.

Activation Record Structure

Each function call creates an activation record containing:

Component	Description	Implementation Reference
Parameters	Function arguments bound to parameter names	callFunction() method
Local Variables	Variables declared within the function	Scope class
Return Value	Storage for the function result	FunctionReturn exception
Control Link	Pointer to the caller's activation record	current_scope.parent reference
Access Link	Pointer to the scope where the function was defined (for closures)	defining_scope in function metadata
Instruction Pointer	The next statement to execute after the function returns	Python call stack

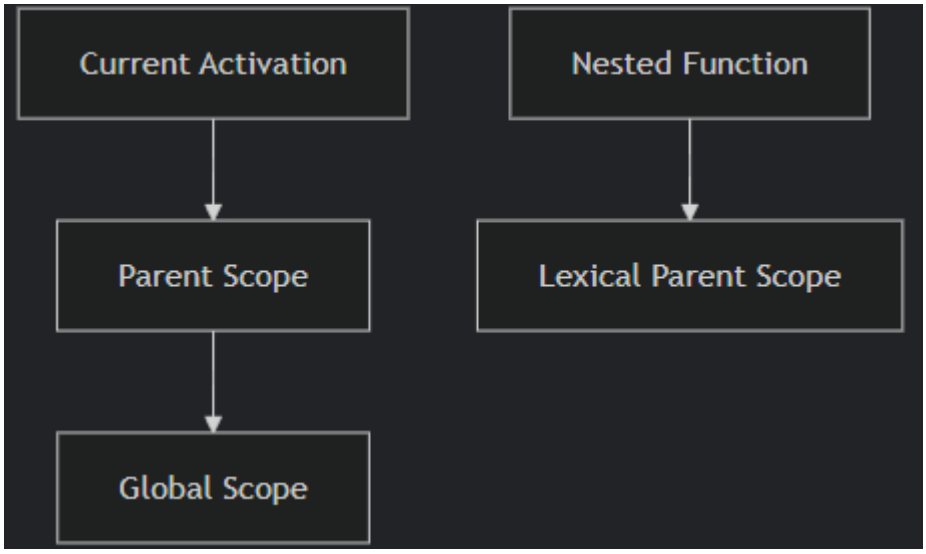
Stack Frame Implementation

The interpreter manages the call stack through these key components:

1. **Call Stack Maintenance**

```
class Interpreter(AingalLangParserVisitor):
    def __init__(self):
        self.call_stack = []  # Tracks active function calls
        self.current_scope = Scope()  # Current activation record

    def callFunction(self, name, args):
        # Create new activation record
        self.call_stack.append({
            'name': name,
            'return_to': len(self.output_lines)
        })

        # Create new scope with access link
        local_scope = Scope(parent=defining_scope)

        # Bind parameters
        for param, arg in zip(params, args):
            local_scope.set_variable(param, arg)

        # Execute function body
        try:
            self.visit(body)
        except FunctionReturn as ret:
            return ret.value
        finally:
            self.call_stack.pop()
```

2. **Variable Access Chain**



Memory Model and Storage

Variable Storage

- Variables are stored in a hierarchical Scope object (Scope.variables).
- Each Scope is a dictionary mapping variable names to **values only**:

```
self.variables = { "x": 100, "msg": "Hello" }
```

- **Type information is discarded** after declaration (values are cast immediately).

Scope Chain

- Each scope may have a parent, allowing parent::x access to outer scopes.
- New scopes are pushed for:
    - Blocks (Block { ... })
    - Loops
    - Function calls

Function Storage

- Functions are stored in self.functions, keyed by name:

```
self.functions["foo"] = {
    "params": ["x", "y"],
    "body": (ANTLR block),
    "scope": (declaration-time scope)
}
```

- **Only parameter names** are stored — not their types.

Runtime Call Stack

- self.current_scope tracks the active scope.
- Function calls push a new local scope, with the function's declaration scope as parent.