README

This is a barebone implementation of GoLang Interpreter in Java. The GoLang package consists of 11 files each of which is described below.

Learnings from the assignment :-

- Application of Visitor Interface in Java.
- Application of HashMap
- Understood ambiguity in grammar and how to resolve it.
-

Environment Class

This Java code defines an `Environment` class as part of the Golang package. The `Environment` class represents an environment for executing a programming language. It is designed to handle variable storage, retrieval, and scoping within an interpreter.

Table of Contents

Overview

The `Environment` class is a fundamental component of an interpreter, providing a mechanism for managing variable bindings and scoping during program execution. It includes features for variable retrieval, assignment, definition, and scoping resolution.

Environment Constructors

Default Constructor

- `Environment()`: Initializes an environment with no enclosing environment.

Parameterized Constructor

- `Environment(Environment enclosing)`: Initializes an environment with the specified enclosing environment.

Environment Methods

Environment Get Method

- `Object get(Token name)`: Retrieves the value associated with the given variable name. If the variable is not found in the current environment, it looks for it in the enclosing environments.

Environment Assign Method

- `void assign(Token name, Object value)`: Assigns a value to the specified variable name. If the variable already exists, it checks for the compatibility of data types before updating the value.

Environment Define Methods

- `void define(String name, Object value)`: Defines a variable with the given name and assigns it the provided value.
- `void define(String name, Object value, int k)`: Defines a variable with the given name, assigns it the provided value, and associates a data type with the variable.

Resolving and Binding Ancestor

- `Environment ancestor(int distance)`: Resolves and returns the ancestor environment at the specified distance.

Resolving and Binding Get-At

- `Object getAt(int distance, String name)`: Retrieves the value associated with the specified variable name in the ancestor environment at the given distance.

Resolving and Binding Assign-At

- `void assignAt(int distance, Token name, Object value)`: Assigns a value to the specified variable name in the ancestor environment at the given distance. It checks for data type compatibility before updating the value.

Utility Methods

- `String stringify(Object object)`: Converts an object to its string representation, handling special cases such as removing trailing ".0" for doubles.
- `int check(Object object)`: Determines the data type of the provided object and returns an integer code (0 for null, 1 for double, 2 for integer, 3 for string, 4 for boolean).

String Representation

- `@Override public String toString()`: Provides a string representation of the environment, including variable bindings and, if applicable, the enclosing environment.

Data Types

The class supports the following data types:

- null
- double
- integer
- string
- boolean

Exception Handling

The class throws a `RuntimeError` if an undefined variable is accessed or if there is an unexpected data type during variable assignment.

Feel free to use this `Environment` class as a foundational component for building an interpreter in Java.

Expr Class Readme

This Java code defines an abstract `Expr` class and its nested subclasses, representing various expressions in the Golang programming language. Each subclass corresponds to a different type of expression, such as literals, unary operations, binary operations, function calls, variable assignments, and more.

Table of Contents

Overview

The `Expr` class serves as the base class for various expression types in the Golang language. Each subclass represents a specific type of expression, encapsulating the structure and behavior associated with that expression.

Expr Subclasses

Expr Assign

- `Assign`: Represents an assignment expression, associating a variable name (`Token`) with an expression (`Expr`) that computes its value.

Expr Binary

- `Binary`: Represents a binary operation expression, involving a left operand (`Expr`), an operator (`Token`), and a right operand (`Expr`).

Expr Call

- `Call`: Represents a function or method call expression, including the callee (`Expr`), the opening parenthesis token (`Token`), and a list of argument expressions (`List<Expr>`).

Expr Get

- `Get`: Represents a property access expression, consisting of an object (`Expr`) and the name of the property (`Token`).

Expr Grouping

- `Grouping`: Represents a grouping expression, enclosing another expression (`Expr`) within parentheses.

Expr Literal

- `Literal`: Represents a literal value expression, such as a number, string, or boolean.

Expr Logical

- `Logical`: Represents a logical operation expression, involving a left operand (`Expr`), a logical operator (`Token`), and a right operand (`Expr`).

Expr Set

- `Set`: Represents a property assignment expression, involving an object (`Expr`), the name of the property (`Token`), and the value to be assigned (`Expr`).

Expr Super

- `Super`: Represents a "super" keyword expression, used to access methods in a superclass.

Expr This

- `This`: Represents a "this" keyword expression, referring to the current instance.

Expr Unary

- `Unary`: Represents a unary operation expression, involving an operator (`Token`) and a right operand (`Expr`).

Expr Variable

- `Variable`: Represents a variable expression, consisting of the variable name (`Token`).

Expr Visitor Interface

The `Visitor` interface defines a set of methods, each corresponding to a specific `Expr` subclass. It allows implementing classes to perform operations on different types of expressions.

Expr Accept Method

The `accept` method in each `Expr` subclass is responsible for invoking the appropriate `Visitor` method based on the specific subclass. This enables the implementation of custom behavior for each type of expression.

Feel free to use these classes as a foundation for building an abstract syntax tree (AST) in Java for the Golang programming language. The provided structure allows for extensibility and flexibility in handling different expression types during interpretation or compilation.

Interpreter Class Readme

This Java code defines an `Interpreter` class responsible for interpreting and executing Golang programs. The interpreter is equipped to handle expressions and statements, including control flow, variable assignments, and function calls. It utilizes an abstract syntax tree (AST) to evaluate and execute the provided Golang code.

Table of Contents

Overview

The `Interpreter` class serves as the core component for interpreting Golang code. It implements both the `Expr.Visitor<Object>` and `Stmt.Visitor<Void>` interfaces to handle expressions and statements, respectively. The interpreter employs an abstract syntax tree (AST) to evaluate expressions and execute statements, allowing for the interpretation of Golang programs.

Interpreter Constructors

Default Constructor

- `Interpreter()`: Initializes a new interpreter.

Interpreter Methods

evaluate

- `public Object evaluate(Expr expr)`: Evaluates the given expression using the visitor pattern.

execute

- `private void execute(Stmt stmt)`: Executes the given statement using the visitor pattern.

resolve

- `void resolve(Expr expr, int depth)`: Resolves the scope of a variable within the interpreter.

executeBlock

- `void executeBlock(List<Stmt> statements, Environment environment)`: Executes a block of statements within the specified environment, handling variable scope.

executeBlock2

- `void executeBlock2(List<Stmt> statements, Environment environment)`: Executes a block of statements within the specified environment without modifying the interpreter's current environment.

Interpreter Visitor Methods

Statements and State

- `visitBlockStmt(Stmt.Block stmt)`: Visits a block statement, executing a block of statements within a new environment.
- `visitBlockStmt2(Stmt.Block2 stmt)`: Visits a block statement without modifying the interpreter's current environment.
- `visitClassStmt(Stmt.Class stmt)`: Visits a class statement.
- `visitExpressionStmt(Stmt.Expression stmt)`: Visits an expression statement, evaluating the expression.
- `visitFunctionStmt(Stmt.Function stmt)`: Visits a function statement.
- `visitIfStmt(Stmt.If stmt)`: Visits an if statement, evaluating the condition and executing the appropriate branch.
- `visitPrintStmt(Stmt.Print stmt)`: Visits a print statement, evaluating the expression and printing the result.
- `visitPrintlnStmt(Stmt.Println stmt)`: Visits a println statement, evaluating the expression and printing the result with a newline.
- `visitReturnStmt(Stmt.Return stmt)`: Visits a return statement.
- `visitVarStmt(Stmt.Var stmt)`: Visits a variable declaration statement.
- `visitVarIntStmt(Stmt.VarInt stmt)`: Visits an integer variable declaration statement.
- `visitVarDoubleStmt(Stmt.VarDouble stmt)`: Visits a double variable declaration statement.
- `visitVarStringStmt(Stmt.VarString stmt)`: Visits a string variable declaration statement.
- `visitVarBoolStmt(Stmt.VarBool stmt)`: Visits a boolean variable declaration statement.
- `visitWhileStmt(Stmt.While stmt)`: Visits a while loop statement, evaluating the condition and executing the body while the condition is true.
- `visitAssignExpr(Expr.Assign expr)`: Visits an assignment expression, evaluating the value and assigning it to a variable.

Functions

- `visitCallExpr(Expr.Call expr)`: Visits a function or method call expression.

Classes

- `visitGetExpr(Expr.Get expr)`: Visits a property access expression.
- `visitSetExpr(Expr.Set expr)`: Visits a property assignment expression.
- `visitSuperExpr(Expr.Super expr)`: Visits a super expression.
- `visitThisExpr(Expr.This expr)`: Visits a this expression.

Control Flow

- `visitLogicalExpr(Expr.Logical expr)`: Visits a logical operation expression.
- `visitIfStmt(Stmt.If stmt)`: Visits an if statement, evaluating the condition and executing the appropriate branch.
- `visitWhileStmt(Stmt.While stmt)`: Visits a while loop statement, evaluating the condition and executing the body while the condition is true.

Utility Methods

- `checkNumberOperand(Token operator, Object operand)`: Checks if the operand is a number; otherwise, throws an error.
- `checkNumberOperands(Token operator, Object left, Object right)`: Checks if both operands are numbers; otherwise, throws an error.
- `isTruthy(Object object)`: Determines if the given object is truthy.
- `isEqual(Object a, Object b)`: Compares two objects for equality.
- `stringify(Object object)`: Converts an object to its string representation.

Resolving and Binding

The interpreter supports resolving variable scope using the `resolve` method. It utilizes a map (`locals`) to associate expressions with their corresponding scope depth.

Exception Handling

The interpreter throws `RuntimeError` exceptions in case of runtime errors, providing details about the error, such as the token involved.

Utility Methods

The interpreter includes utility methods for checking operands, determining truthiness, comparing equality, and converting objects to strings.

Feel free to use this `Interpreter` class as a foundation for building an interpreter for the Golang programming language in Java.

GoLangInstance Class Readme

This Java code defines a `GoLangInstance` class that represents instances of user-defined classes in the Golang programming language. Each instance maintains a map of fields, allowing for the storage and retrieval of properties. This README provides an overview of the `GoLangInstance` class, its methods, and how it can be used.

Table of Contents

Overview

The `GoLangInstance` class is designed to represent instances of user-defined classes in Golang. It encapsulates a map (`fields`) to store and retrieve properties dynamically. Instances of this class can be used to model objects with properties, enabling property access and modification.

GoLangInstance Methods

get

- `Object get(Token name)`: Retrieves the value of a property with the specified name from the instance's fields. If the property is not found, a `RuntimeError` is thrown, indicating that the property is undefined.

set

- `void set(Token name, Object value)`: Sets the value of a property with the specified name in the instance's fields. This method allows for dynamic addition or modification of properties.

Main Class Readme

This Java code implements an interpreter for the Golang programming language. The interpreter is capable of parsing and executing Golang source code, reporting errors during the process. This README provides an overview of the main components and functionalities of the interpreter.

Table of Contents

Overview

The Golang interpreter is designed to parse and execute Golang source code. It utilizes a scanner to tokenize the input, a parser to build an abstract syntax tree (AST), and an interpreter to evaluate and execute the AST.

Usage

//this needs to be written

Code Structure

Main Class

The `Main` class serves as the entry point for the interpreter. It handles command-line arguments, runs scripts, and provides an interactive mode (REPL) for running Golang code.

Interpreter Class

The `Interpreter` class implements the logic for evaluating Golang expressions and executing statements. It utilizes a scanner to tokenize the source code, a parser to build the AST, and a resolver for variable resolution.

Scanner Class

The `Scanner` class is responsible for tokenizing Golang source code. It scans the input and produces a list of tokens that represent the lexical elements of the code.

Parser Class

The `Parser` class parses the list of tokens generated by the scanner and builds an abstract syntax tree (AST). It is responsible for handling the syntax of Golang expressions and statements.

Error Handling

The interpreter includes error handling mechanisms to report syntax errors, runtime errors, and other issues. The `Main` class contains functions for reporting errors, and error flags (`hadError` and `hadRuntimeError`) are used to control the program's behavior in case of errors.

Feel free to explore and extend this Golang interpreter codebase to enhance its features or integrate additional functionalities.

GoLang Parser Readme

This Java code implements a parser for the Golang programming language. The parser is designed to process tokens generated by the scanner and build an abstract syntax tree (AST). This README provides an overview of the main components and functionalities of the parser.

Table of Contents

Overview

The Golang parser is responsible for processing Golang source code and converting it into a structured abstract syntax tree (AST). It handles various language constructs such as declarations, statements, and expressions, ensuring that the input adheres to the Golang syntax.

Code Structure

Parser Class

The `Parser` class contains methods to parse different elements of Golang code, including declarations, statements, and expressions. It uses recursive descent parsing to handle various grammar rules.

- **Parsing Declarations:** The parser supports the parsing of variable declarations, including data type specifications and initializations.
- **Parsing Statements:** It handles various types of statements, such as `if`, `for`, and block statements. Additionally, it parses print statements and handles library imports.
- **Parsing Expressions:** The parser processes expressions involving logical and arithmetic operations, as well as function calls and variable references.

Usage

Parsing Declarations

To parse variable declarations, the parser provides methods such as `varDeclaration()` and `varDeclarationDataType()`. These methods handle variable declarations with and without explicit data types.

Parsing Statements

The parser supports the parsing of various statements, including `if` statements (`ifStatement()`), `for` statements (`forStatement()`), and block statements (`block()`). Additionally, it handles print statements (`printlnStatement()` and `printStatement()`).

Parsing Expressions

The parser processes Golang expressions using methods such as `expression()`, `assignment()`, `or()`, `and()`, `equality()`, `comparison()`, `term()`, `factor()`, `unary()`, `primary()`, `call()`, and `finishCall()`.

Error Handling

The parser includes error-handling mechanisms to report syntax errors and recover from them. The `error()` method is used to report errors, and the `synchronize()` method helps recover from errors and continue parsing.

Feel free to explore and modify this Golang parser code to suit your needs or extend its functionality. Understanding the structure of the parser can help you enhance or adapt it for specific language features or requirements.

Golang Runtime Error Class

Overview

The `RuntimeError` class in the Golang codebase is a custom exception class designed to handle runtime errors encountered during the execution of Golang code. This class extends the built-in `RuntimeException` class in Java, providing additional functionality specific to Golang error handling.

Usage

Constructor

java
```
RuntimeError(Token token, String message)
```

- **Parameters:**

    - `Token token`: Represents the token where the error occurred.
    - `String message`: Describes the nature of the runtime error.
- **Description:**
    - Initializes the `token` field with the provided token.
    - Calls the constructor of the superclass (`RuntimeException`) with the given error message.

In the example above, if a runtime error occurs, it is caught as a `RuntimeError`. The associated token and error message can be accessed to provide detailed information about the error, such as the line where it occurred.

Integration

To integrate this `RuntimeError` class into the Golang interpreter, follow these steps:

1. **Include in Golang Package:**
    - Make sure the `RuntimeError` class is part of the `Golang` package.
2. **Throwing Runtime Errors:**
    - Identify points in the Golang interpreter where runtime errors may occur.
    - Instantiate a `RuntimeError` object with the appropriate token and error message.
    - Throw the `RuntimeError` to signal and handle the runtime error.
3. **Error Handling:**
    - Implement error-handling logic to catch and handle instances of `RuntimeError`.
    - Extract information from the caught exception, such as the token and error message, to provide meaningful feedback to the user.

Golang Scanner

Overview

The `Scanner` class in the Golang codebase is responsible for lexical analysis, breaking down the source code into a sequence of tokens. Tokens are the smallest units of meaning in a programming language and are used by the interpreter to understand and execute the code.

Usage

Constructor

java
```
Scanner(String source)
```

- **Parameters:**

    - `String source`: The source code to be scanned.

Public Method

java
```
List<Token> scanTokens()
```

- **Returns:**
    - A list of `Token` objects representing the scanned tokens.
- **Description:**
    - Initiates the process of scanning tokens from the source code.
    - Invokes the `scanToken` method to scan individual tokens until the end of the source code is reached.
    - Appends an `EOF` (end-of-file) token to mark the completion of the scanning process.

Token Types

The scanner recognizes various token types, including:

- **Single-character tokens:**
    - `(`, `)`, `{`, `}`, `,`, `.`, `-`, `+`, `;`, `*`, `/`
- **Two-character tokens:**
    - `!=`, `==`, `:=`, `<`, `<=`, `>`, `>=`
- **Keywords:**
    - `and`, `class`, `else`, `false`, `for`, `func`, `if`, `nil`, `or`, `fmt`, `Println`, `Print`, `return`, `true`, `var`, `int`, `float`, `bool`, `string`, `main()`, `import`
- **Literals:**
    - Numbers (e.g., `123`, `3.14`)
    - Strings (e.g., `"Hello, World!"`)
- **Identifiers:**
    - Variable and function names (e.g., `variable`, `function()`)

In this example, the source code is scanned, and the resulting tokens are printed. This helps in debugging and understanding how the scanner breaks down the code.

Integration

To integrate this `Scanner` class into the Golang interpreter, follow these steps:

1. **Instantiate the Scanner:**
    - Create an instance of the `Scanner` class, providing the source code as a parameter.
2. **Scan Tokens:**
    - Call the `scanTokens` method to obtain a list of tokens.
3. **Token Processing:**
    - Utilize the list of tokens for further processing, such as parsing and interpreting the Golang code.

Golang Statement (Stmt) Abstract Class

Overview

The `Stmt` (Statement) class in the Golang codebase represents abstract syntax tree nodes for various statements in the programming language. Statements are fundamental building blocks that define the actions or operations to be performed.

Statement Types

The `Stmt` class includes several nested classes, each representing a specific type of statement:

1. **Block (`Block` and `Block2`):**
    - Represents a block of statements enclosed within `{}`.
2. **Class (`Class`):**

    - Represents a class declaration, including its name, superclass (if any), and methods.
3. **Expression (`Expression`):**
    - Represents an expression statement, which evaluates an expression without assigning it to a variable.
4. **Function (`Function`):**
    - Represents a function or method declaration, including its name, parameters, and body.
5. **If (`If`):**
    - Represents an if statement with a condition, a branch to execute if the condition is true, and an optional else branch.
6. **Print (`Print` and `Println`):**
    - Represents a print statement, which outputs the result of an expression to the console.
7. **Return (`Return`):**
    - Represents a return statement inside a function, including the keyword and the value to be returned.
8. **Variable (`Var`, `VarInt`, `VarDouble`, `VarString`, `VarBool`):**
    - Represents variable declarations with an optional initializer.
9. **While (`While`):**
    - Represents a while loop with a condition and a body.

Visitor Pattern

The `Stmt` class uses the visitor pattern, allowing external classes to traverse and perform operations on the statement nodes. The `Visitor` interface defines methods corresponding to each statement type.

Integration

To integrate the `Stmt` class into the Golang interpreter, follow these steps:

1. **Instantiate Statements:**
    - Create instances of the appropriate nested classes to represent the statements in the source code.
2. **Visitor Implementation:**
    - Implement the `Visitor` interface to define actions for each statement type.
3. **Accept Method:**

    - Implement the `accept` method in each statement class to invoke the corresponding `Visitor` method.
4. **Traversal:**
    - Traverse the abstract syntax tree using the visitor pattern to perform operations or generate code.

Golang Token Class

Overview

The `Token` class in the Golang codebase is responsible for representing lexical tokens generated during the scanning phase of the interpreter. Tokens are fundamental units of the source code and play a crucial role in the parsing and interpretation process.

Token Structure

The `Token` class encapsulates the following information for each token:

- **Type (`type`):** Enumerates the type of the token (e.g., keyword, identifier, number, etc.).
- **Lexeme (`lexeme`):** Represents the actual text of the token as found in the source code.

- **Literal (`literal`):** Holds the literal value associated with the token, if applicable (e.g., the numeric value for a number token).
- **Line (`line`):** Indicates the line number in the source code where the token was found, aiding in error reporting and debugging.

Token Types

The `TokenType` enum, which is used in conjunction with the `Token` class, defines a set of constants representing different types of tokens recognized by the Golang interpreter. These may include keywords, identifiers, literals, and special symbols.

Integration

To integrate the `Token` class into the Golang interpreter, follow these steps:

1. **Scanning:**
    - Utilize the `Token` class during the scanning phase to generate tokens while reading the source code.
2. **Token Types:**
    - Leverage the `TokenType` enum to categorize tokens into different types.
3. **Error Handling:**
    - Utilize the line information in the `Token` class for effective error reporting during parsing and interpretation.
4. **Parsing:**
    - Pass the generated tokens to the parser to construct the abstract syntax tree.

Golang Token Types

Overview

The `TokenType` enum in the Golang codebase defines a comprehensive set of token types used during the scanning phase of the interpreter. Tokens are essential building blocks in the interpretation process, representing the different syntactic elements of the source code.

Token Categories

Tokens are categorized into several groups, each representing a specific syntactic construct. The categories include:

Single-Character Tokens

- **Left Parenthesis (`LEFT_PAREN`):** "("
- **Right Parenthesis (`RIGHT_PAREN`):** ")"
- **Left Brace (`LEFT_BRACE`):** "{"
- **Right Brace (`RIGHT_BRACE`):** "}"
- **Comma (`COMMA`):** ","
- **Dot (`DOT`):** "."
- **Minus (`MINUS`):** "-"
- **Plus (`PLUS`):** "+"
- **Semicolon (`SEMICOLON`):** ";"
- **Slash (`SLASH`):** "/"
- **Star (`STAR`):** "*"

One or Two Character Tokens

- **Bang (`BANG`):** "!"
- **Bang Equal (`BANG_EQUAL`):** "!="
- **Equal (`EQUAL`):** "="
- **Equal Equal (`EQUAL_EQUAL`):** "=="
- **Greater (`GREATER`):** ">"
- **Greater Equal (`GREATER_EQUAL`):** ">="
- **Less (`LESS`):** "<"
- **Less Equal (`LESS_EQUAL`):** "<="

Literals

- **Identifier (`IDENTIFIER`):** Represents variable names and identifiers.
- **String (`STRING`):** Represents string literals.
- **Number (`NUMBER`):** Represents numeric literals.

Keywords

- **AND, CLASS, ELSE, FALSE, FUN, FOR, IF, NIL, OR, PRINT, RETURN, SUPER, THIS, TRUE, VAR, WHILE:** Keywords representing various language constructs.
- **COLON_EQUAL (`COLON_EQUAL`):** ":="
- **INT, FLOAT, BOOL, PRINTLN, STR:** Additional keyword types.
- **ENDL:** Represents the end of a line.
- **FMT:** Represents the "fmt" keyword.
- **MAIN, IMPORT:** Keywords for main function and import statements.

Special Tokens

- **EOF:** Represents the end of the file.

Integration

To use the `TokenType` enum in the Golang interpreter, follow these steps:

1. **Scanning:**
    - Utilize the `TokenType` enum to categorize tokens generated during the scanning phase.
2. **Token Handling:**
    - Associate each token with its corresponding `TokenType` during token creation.
3. **Parsing:**
    - Leverage the token types to guide the parser in constructing the abstract syntax tree.
4. **Language Constructs:**
    - Understand the significance of each token type in the context of Golang language constructs.

Contributions

Anuj Sharma : For loop, Variable declaration, Arithmetic operations, Interpreting statements.
Sai Madhav : Variable declaration, Print statements, Interpreting statements.
Ashish Raj : Variable scope, blocks, Parsing expressions.
Preet Bobde : Errors display and exception handling, Readme file, Parsing expressions
Jyothiraditya : If else statement, Readme file, Parsing expressionss