Project Part 3: Interpreter

New Attempt

Due Mar 17 by 11:59pm **Points** 100 **Submitting** a file upload **File Types** java

In this assignment, you will implement the interpreter for our language. The interpreter is responsible for evaluating code, aka the AST, which we will do using the visitor pattern. We have provided a few files for managing objects and scope; your focus will be on implementing the evaluation of the different `Ast` classes in the interpreter itself.

Submission

You will submit `Interpreter.java`, which implements your interpreter. The provided classes should remain the same, and you do not need to submit any additional test cases you have added. You should retest your interpreter with the provided test class prior to submitting.

- There will be a test submission on **Friday, March 12** and **Monday, March 15**
- The final submission is **Wednesday, March 17**

Interpreter Overview

The interpreter is responsible for evaluating code - the AST. To do this, will use the visitor pattern where each `visit` method evaluates the corresponding `Ast` class, returning the appropriate value (if any). Our interpreter is going to be dynamic, and so we will need to take into accout the possibility for failures like undefined variables or invalid types.

For example, evaluating the AST represented by `1 + 2 * 3` returns the runtime value of the number `7`. The wording here is important - the runtime value of `7` is a `PlcObject` containing a `BigInteger` as opposed to just an `int` or any combination. Values in the specification will generally be represented as unwrapped values, so `nil` means `Environment.NIL`.

Managing Scope

Scopes are used to track what functions/variables an identifier refers to. The same name can be used in different scopes (such as a class field and function parameter), and thus scopes are represented using a hierarchy to allow this. The implementation of the `Scope` class is provided, but you need to have an understanding of how it works to use it properly.

For example, in Java we could have code which looks like this:

```
int x = 0;
if (true) {
    x = 1;
    int y = 2;
}
```

After evaluating this, `x` is either `0` or `1` and `y` is... `2`? Actually, after this if statement `y` isn't anything - it's simply undefined - and this is because it was defined within the scope of the if statement which has finished. On the other hand, the value of `x` has indeed changed, since it is not defined in the scope of the if statement but is accessible from it.

Additionally, because we have the possibility of exceptions being thrown for returning values, we need to restore the scope in a `finally` block to ensure it is run. See lecture notes for details.

Error Handling

If evaluation fails, the interpreter should throw a normal `RuntimeException`. Previously we used a custom `ParseException` because we had index information, but we don't have anything special in the interpreter (ideally we would things like line numbers and stacktraces, but easier said than done).

Note that many of the provided classes and helper functions already handle some error conditions - for example, the `scope.lookup` methods handle the case of the variable/function being undefined. Likewise, `requireType` also has error handling covered.

Crafting Interpreters

The following sections of Crafting Interpreters will be helpful for implementing the interpreter, and covers nearly all of the important parts of this project. However, remember that this is a different language and so some behavior may be different.

- Evaluating Expressions
- Statements and State
- Control Flow
- Functions

Ast Types

Each `Ast` class has it's own `visit` method, which behave as follows:

AST Class	Specification	Examples

`Ast.Source`	Evalutes fields followed by methods. Returns the result of calling the `main/0` function (named `main` with arity `0`). If this function doesn't exist an error should be thrown.	• `DEF main() DO RETURN 0; END`
		  ◦ `0`
		• `LET x = 1; LET y = 10; DEF main() DO x + y END`
		  ◦ `NIL` (`x + y` not returned)

`Ast.Field`	Defines a variable in the current scope, defaulting to `NIL` if no initial value is defined. Returns `NIL`.	• `LET name;`, scope = `{}`
		  ◦ `NIL`
		  ◦ scope = `{name = NIL}`
		• `LET name = 1`, scope = `{}`
		  ◦ `NIL`
		  ◦ scope = `{name = 1}`

`Ast.Method`	Defines a function in the current scope. The callback function (lambda) should implement the behavior of calling this method, which needs to do the following:
	• Set the scope to be a new child of the scope *where the function was defined* (hint: you need to capture this in a variable).
	  ◦ Remember to restore the scope when finished!
	• Define variables for the incoming arguments, using the parameter names. You may assume the right number of arguments are provided (since the arity is checked else).
	• Evaluate the methods statements. Returns the value contained in a `Return` exception if thrown, otherwise `NIL`.
	Finally, `visit(Ast.Method)` should itself return `NIL`. Note that there are two levels of return here, one in `visit` and the other within the lambda expression.
	Remember there is an example of defining the `print` function in the constructor of `Interpreter`.
		• `DEF main() DO RETURN 0; END`
		  ◦ `NIL`
		  ◦ scope = `{main = ...}`, where evaluating `main()` return `0`
		• `DEF square(x) DO RETURN x * x END`
		  ◦ `NIL`
		  ◦ scope = `{square = ...}`, where evaluating `square(10)` returns `100`

`Ast.Stmt.Expression`	Evaluates the expression. Returns `NIL`.	• `print("Hello, World!");`
		  ◦ `NIL`
		  ◦ prints `Hello, World!`

`Ast.Stmt.Declaration`	Defines a variable in the current scope, defaulting to `NIL` if no initial value is defined. Returns `NIL`.
	Yes, this indeed does the same thing as `Ast.Field`.
		• `LET name;`, scope = `{}`
		  ◦ `NIL`
		  ◦ scope = `{name = NIL}`
		• `LET name = 1`, scope = `{}`
		  ◦ `NIL`
		  ◦ scope = `{name = 1}`

`Ast.Stmt.Assignment`	First, ensure that the receiver is an `Ast.Expr.Access` (any other type is not assignable). If that access expression has a receiver, evaluate it and set a field, otherwise lookup and set a variable in the current scope. Returns `NIL`.	• `variable = 1;`, scope = `{variable = "variable"}`
		  ◦ `NIL`
		  ◦ scope = `{variable = 1}`
		• `object.field = 1`, scope = `{object = {field = "object.field"}}`
		  ◦ `NIL`
		  ◦ scope = `{object = {field = 1}}`

`Ast.Stmt.If`	Ensure the condition evaluates to a `Boolean` (hint: use `requireType`). Inside of a new scope, if the condition is `TRUE`, evalute `thenStatements`, otherwise evalute `elseStatements`. Returns `NIL`.	• `IF TRUE DO num = 1; END`, scope = `{num = NIL}`
		  ◦ `NIL`
		  ◦ scope = `{num = 1}`
		• `IF FALSE DO ELSE num = 10 END`, scope = `{num = NIL}`
		  ◦ `NIL`
		  ◦ scope = `{num = 10}`

`Ast.Stmt.For`	Ensure the value evaluates to an `Iterable` (hint: use `requireType`). You can assume the contents of the `Iterable` are `PlcObject`s. For each element, inside of a new scope, define a variable with the for loop's name and evaluate the statements. Returns `NIL`.	• `FOR num IN list DO sum = sum + num; END`, scope = `{sum = 0, list = [0, 1, 2, 3, 4]}`
		  ◦ `NIL`
		  ◦ scope = `{sum = 10, list = [0, 1, 2, 3, 4]}`
		  ▪ Note: `num` is not in scope!

`Ast.Stmt.While`	Ensure the condition evaluates to a `Boolean` (hint: use `requireType`). If the condition is `TRUE`, evaluate the statements and repeat. Returns `NIL`.
	• Remember to re-evaluate the condition itself each iteration!
		• `WHILE num < 10 DO num = num + 1; END`, scope = `{num = 0}`
		  ◦ `NIL`
		  ◦ scope = `{num = 10}`

`Ast.Stmt.Return`	Evaluates the value and throws it inside in a `Return` exception (defined at the bottom of `Interpreter.java`).
	• The implementation of `Ast.Method` will catch any `Return` exceptions and complete the behavior.
		• `RETURN 1;`
		  ◦ throws `Return` exception with `value = 1`
		  ◦ Note: `Return` is private in `Interpreter`, must change visibility to include in tests.

`Ast.Expr.Literal`	Returns the literal value as a `PlcObject` (hint: use `Environment.create` as needed).	• `NIL`
		  ◦ `NIL`
		• `1`
		  ◦ `1`

`Ast.Expr.Group`	Evaluates the contained expression, returning it's value.	• `(1)`
		  ◦ `1`
		• `(1 + 10)`
		  ◦ `11`

`Ast.Expr.Binary`	Evaluates arguments based on the specific binary operator, returning the appropriate result for the operation (hint: use `requireType` and `Environment.create` as needed).
	• `AND` / `OR`:
	  ◦ Evaluate the left-hand operand, which must be a `Boolean`. Following short circuiting rules, evaluate the right-hand operand, which also must be a `Boolean`, if necessary.
	• `<` / `<=` / `>` / `>=`:
	  ◦ Evaluate the left-hand operand, which must be `Comparable`, and compare it to the right-hand operator, which must be *the same type (class) as the left-hand operator*.
	  ◦ You will need to determine how to use `Comparable`.
	• `==` / `!=`:
	  ◦ Evaluate both operands and test for equality using `Objects.equals` (this is **not** the standard equals method).
	• `+`:
	  ◦ Evaluate both operands. If **either** operand is a `String`, the result is their concatenation. Else, if the left-hand operator is a `BigInteger`/`BigDecimal`, then the right-hand operator must also be the same type (a `BigInteger`/`BigDecimal`) and the result is their addition, otherwise throw an error.
	• `-` / `*`:
	  ◦ Evaluate both operands. If the left-hand operator is a `BigInteger`/`BigDecimal`, then the right-hand operator must also be the same type (a `BigInteger`/`BigDecimal`) and the result is their subtraction/multiplication, otherwise throw an error.
	• `/`:
	  ◦ Evaluate both operands. If the left-hand operator is a `BigInteger`/`BigDecimal`, then the right-hand operator must also be the same type (a `BigInteger`/`BigDecimal`) and the result is their division, otherwise throw an error.
	  ◦ For `BigDecimal`, use `RoundingMode.HALF_EVEN`, which rounds midpoints to the nearest even value (`1.5`, `2.5` -> `2.0`). This is actually the default mode in Python, which can often catch developers off-guard who aren't expecting it.
	  ◦ If the denominator is zero, throw an error.
		• `TRUE AND FALSE`
		  ◦ `FALSE`
		• `TRUE OR undefined`
		  ◦ `TRUE` (without visiting `undefined`)
		• `1 < 10`
		  ◦ `TRUE`
		• `1 >= 10`
		  ◦ `FALSE`
		• `1 == 10`
		  ◦ `FALSE`
		• `"a" + "b"`
		  ◦ `"ab"`
		• `1 + 10`
		  ◦ `11`
		• `1.2 / 3.4`
		  ◦ `0.4`

`Ast.Expr.Access`	If the expression has a receiver, evaluate it and return the value of the appropriate field, otherwise return the value of the appropriate variable in the current scope.	• `variable`, scope = `{variable = "variable"}`
		  ◦ `"variable"`
		• `object.field`, scope = `{object = {field = "object.field"}}`
		  ◦ `"object.field"`

`Ast.Expr.Function`	If the expression has a receiver, evaluate it and return the result of calling the appropriate method, otherwise return the value of invoking the appropriate function in the current scope with the evaluated arguments.	• `function()`, scope = `{function = ...}` where `function` takes no arguments and returns `"function"`
		  ◦ `"function"`
		• `object.method()`, scope = `{object = {method = ...}}` where `method` takes no arguments and returns `"object.method"`
		  ◦ `"object.method"`

Provided Code

The following files are provided to help you help implement the interpreter.

- Source Files (`src/main/java/plc/project`)
  - Ast.java
  - Environment.java
  - Interpreter.java
  - Scope.java
- Test Files (`src/test/java/plc/project`)
  - InterpreterTests.java