Project Part 4: Analyzer

New Attempt

Due Apr 9 by 11:59pm **Points** 100 **Submitting** a file upload **File Types** java

In this assignment, you will implement the analyzer for our language. This is the third step in the compiler process which takes the AST produced by the parser and updates it with attribute data that will help perform code generation in the next project part.

Submission

You will submit `Analyzer.java`; your updated `Parser.java` is not graded as part of the analyzer and will not be submitted until the final evaluation (thus, giving you time to work on the rest of the parser if you still need to). You do not need to submit any additional tests you have created (though you should absolutely use them to your advantage).

- There will be two test submissions, one on **Thursday, April 1** and another on **Monday, April 5**.
- The final submission is **Friday, April 9**.

Semantic Analysis Overview

Recall that the job of semantic analysis is determine if the semantic structure of the tokens and AST matches with the programming language.

Operations to perform are:

- The AST will be evaluated for consistency and constraints. Examples (see below for the section Ast Types For Semantic Analysis and complete details):
  - Empty statement bodies of the for statement.
  - Assignment Receiver must be an Access expression.
- Within the AST, type information will be attached to the AST node, providing type information to type validation (here in Part 4) and the Generator in Part 5.
- Type and variable validation using the structure provided (including continuing to use `Environment.java` and `Scope.java`.

Enhancement to the Grammar

In order for our grammar to include variable types, the declaration of Fields and Method parameters must include the type. The enhanced Parser will check for the types being present and throw an exception if they are not.

The declaration of (non-Field) variables must include the type or an assignment so that the type can be inferred from the assigned value. Evaluation of the type or literal assignment (or both) being present will be left to the Analyzer. Since both the type name and assigned value are optional, the parser will populate one or both when they are present in the source code. However, the Parser will not evaluate if both the type and the assigned value are missing. Instead, the Analyzer will check for these and if they are both missing throw an exception when visiting the declaration node of the tree.

Note, a type is simply an *identifier*. We will use a preceding *colon*, ':', to separate the type from the name of the variable. Both of these elements, an *identifier* and a *colon*, are already lexed correctly by our Lexer. Therefore, no update to the Lexer is required. The type will need to be added into our AST by the Parser. Three productions have changed (field, method, and statement) with new elements highlighted in red:

- `field ::= 'LET' identifier ':' identifier ('=' expression)? ';'`
- `method ::= 'DEF' identifier '(' (identifier ':' identifier (',' identifier ':' identifier)* )? ')' (':' identifier)? 'DO' statement* 'END'`
- `statement ::= 'LET' identifier (':' identifier)? ('=' expression)? ';' |`

Here is the complete grammar with new elements highlighted in red. You can view a graphical form of our grammar on the following website:

- https://www.bottlecaps.de/rr/ui

```
source ::= field* method*
field ::= 'LET' identifier ':' identifier ('=' expression)? ';'
method ::= 'DEF' identifier '(' (identifier ':' identifier (',' identifier ':' identifier)* )? ')' (':' identifier)? 'DO' statement* 'END'

statement ::=
    'LET' identifier (':' identifier)? ('=' expression)? ';' |
    'IF' expression 'DO' statement* ('ELSE' statement*)? 'END' |
    'FOR' identifier 'IN' expression 'DO' statement* 'END' |
    'WHILE' expression 'DO' statement* 'END' |
    'RETURN' expression ';' |
    expression ('=' expression)? ';'

expression ::= logical_expression

logical_expression ::= comparison_expression (('AND' | 'OR') comparison_expression)*
comparison_expression ::= additive_expression (('<' | '<=' | '>' | '>=' | '==' | '!=') additive_expression)*
additive_expression ::= multiplicative_expression (('+' | '-') multiplicative_expression)*
multiplicative_expression ::= secondary_expression (('*' | '/') secondary_expression)*

secondary_expression ::= primary_expression ('.' identifier ('(' (expression (',' expression)*)? ')')?)*

primary_expression ::=
    'NIL' | 'TRUE' | 'FALSE' |
    integer | decimal | character | string |
    '(' expression ')' |
    identifier ('(' (expression (',' expression)*)? ')')?

identifier ::= [A-Za-z_] [A-Za-z0-9_]*
number ::= [+-]? [0-9]+ ('.' [0-9]+)?
character ::= ['] ([^'\\] | escape) [']
string ::= '"' ([^"\n\r\\] | escape)* '"'
escape ::= '\' [bnrt'"\]
operator ::= [<>!=] '='? | 'any character'

whitespace ::= [ \b\n\r\t]
```

AST Representation for Parsing Variable Types

The `Ast` class contains subclasses representing the more specific elements of the AST and the code that is being parsed. There is not a one-to-one relationship between rules in our grammar and the AST.

For our language, the names and spelling of our types will be:

- Boolean
- Character
- Decimal
- Integer
- String

The identifier given within the source at the appropriate position (after the colon and prior to the optional assignment) is the type. No type checking (verification) is performed by the Parser, making the Parser step merely just a recording of the String value of the type name. Type checking will be performed by the Analyzer.

These classes within `Ast` have been updated to represent variable types:

- `Field`: Field declarations now include an attribute **typeName**.
- `Method`: Method definitions now include attributes **parameterTypeNames** (a list of Strings, corresponding to the order of the types given in the parameter list of the method) and **returnTypeName**.
- `Stmt`: Structural parts of the code that perform side effects like assigning variables or modifying control flow.
  - `Declaration`: Variable declarations now include an attribute **typeName**.

Assignable Types

A type is assignable to another if it is a subtype, such as `String` and `Object` in Java (corresponding to `Object obj = "string";`). In our language, we do not have inheritance and instead use the following rules to determine if a type is assignable to another.

- When the two types are the same, the assignment can be performed.
- When the target type is `Any`, anything from our language can be assigned to it. `Any` in our language is similar to the `Object` class in Java.
- When the target type is `Comparable`, it can be assigned any of our defined `Comparable` types: `Integer`, `Decimal`, `Character`, and `String`. You do not need to support any other types.
- In all other cases, an assignment will fail throwing a `RuntimeException`.

Implement and use `requireAssignable(Environment.Type target, Environment.Type type)` in `Analyzer.java` to identify RuntimeException should be thrown when the target type does not match the type being used or assigned. Note, the method returns void because either the exception is generated or the requirement is met.

Ast Types for Semantic Analysis

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

AST Class	Specification	Examples
`Ast.Source`	Visits fields followed by methods. Throws a `RuntimeException` if: • A `main/0` function (name = `main`, arity = `0`) does not exist. • The `main/0` function does not have an `Integer` return type. Returns `null`.	• `LET num: Integer = 1; DEF main(): Integer DO print(num + 1.0); END` ◦ throws `RuntimeException` • `DEF main() DO print("Hello, World!"); END` ◦ throws `RuntimeException`
`Ast.Field`	Defines a variable in the current scope according to the following, also setting it in the Ast (`Ast.Field#setVariable`). • The variable's `name` and `jvmName` are both the name of the field. • The variable's `type` is the type registered in the `Environment` with the same name as the one in the AST. • The variable's `value` is `Environment.NIL` (since it is not used by the analyzer) The value of the field, if present, must be visited *before* the variable is defined (otherwise, the field would be used before it was initialized). Additionally, throws a `RuntimeException` if: • The value, if present, is not assignable to the field. ◦ For a value to be assignable, it's type must be a subtype of the field's type as defined above. Returns `null`.	• `LET name: Integer;`, scope = `{}` ◦ scope = `{name: Integer}` • `LET name: Decimal = 1;`, scope = `{}` ◦ throws `RuntimeException` • `LET name: Unknown;`, scope = `{}` ◦ throws `RuntimeException`
`Ast.Method`	Defines a function in the current scope according to the following, also setting it in the Ast (`Ast.Method#setFunction`). • The function's `name` and `jvmName` are both the name of the method. • The function's parameter types and return type are retrieved from the environment using the corresponding names in the method. If the return type is not present in the AST, it should be `Nil`. • The function's `function` (such naming much wow) is `args -> Environment.NIL`, which always returns nil (since it is not used by the analyzer). Then, visits all of the method's statements inside of a new scope containing variables for each parameter. Unlike fields, this is done *after* the function is defined to allow for recursive functions. Additionally, you will need to somehow coordinate with `Ast.Stmt.Return` so the expected return type is known (hint: save in a variable). • Note: You do **NOT** need to check for missing returns or 'dead' code (statements after a return), both of which are errors in Java. ◦ Consider offering bonus points for this? Returns `null`.	• `DEF main(): Integer DO RETURN 0; END`, scope = `{}` ◦ scope = `{main/0: () -> Integer}` • `DEF increment(num: Integer): Decimal DO RETURN num + 1; END` ◦ throws `RuntimeException`
`Ast.Stmt.Expression`	Validates the expression statement. Throws a `RuntimeException` if: • The expression is not an `Ast.Expr.Function` (since this is the only type of expression that can cause a side effect). Returns `null`.	• `print(1);` ◦ success • `1;` ◦ throws `RuntimeException`
`Ast.Stmt.Declaration`	Defines a variable in the current scope according to the following: • The variable's `name` and `jvmName` are both the name in the AST. • The variable's `type` is the type registered in the `Environment` with the same name as the one in the AST, if present, or else the type of the value. If neither are present this is an error. • The variable's `value` is `Environment.NIL` (since it is not used by the analyzer). The value of the field, if present, must be visited *before* the variable is defined (otherwise, the field would be used before it was initialized and also because it's type may be needed to determine the type of the variable). Additionally, throws a `RuntimeException` if: • The value, if present, is not assignable to the variable (see `Ast.Field` for info). Returns `null`.	• `LET name: Integer;`, scope = `{}` ◦ scope = `{name: Integer}` • `LET name = 1;`, scope = `{}` ◦ scope = `{name: Integer}` • `LET name;` ◦ throws `RuntimeException` • `LET name: Unknown;` ◦ throws `RuntimeException`
`Ast.Stmt.Assignment`	Validates an assignment statement. Throws a `RuntimeException` if: • The receiver is not an access expression (since any other type is not assignable). • The value is not assignable to the receiver (see `Ast.Field` for info). In the interpreter, we had to do attional work to unwrap names in the AST. Here, we do **not** need to do that since visitng the AST is performing type analysis, not evaluation, and thus the behaviors are different.	• `variable = 1;`, scope = `{variable: Integer}` ◦ success • `variable = "string"`, scope = `{variable: Integer}` ◦ throws `RuntimeException` • `object.field = 1`, scope = `{object: ObjectType {field: Integer}}` ◦ success
`Ast.Stmt.If`	Validates an if statement. Throws a `RuntimeException` if: • The condition is not of type `Boolean`. • The `thenStatements` list is empty. After handling the condition, visit the then and else statements inside of a new scope *for each one*. Returns `null`.	• `IF TRUE DO print(1); END` ◦ success • `IF "FALSE" DO print(1); END` ◦ throws `RuntimeException` • `IF TRUE DO print(9223372036854775807); END` ◦ throws `RuntimeException` • `IF TRUE DO END` ◦ throws `RuntimeException`
`Ast.Stmt.For`	Validates a for statement. Throws a `RuntimeException` if: • The value is not of type `IntegerIterable`. • The `statements` list is empty. Then, visits all of the for loop's statements in a new scope. This scope should have a variable defined as follows: • The variable's `name` and `jvmName` are both the name in the AST. • The variable's `type` is `Integer`. • The variable's `value` is `Environment.NIL` (since it is not used by the analyzer). Returns `null`.	• `FOR num IN list DO print(num); END`, scope = `{list: IntegerIterable}` ◦ success
`Ast.Stmt.While`	Validates a while statement. Throws a `RuntimeException` if: • The value is not of type `Boolean`. Then, visits all of the while loop's statements in a new scope. Returns `null`.	`WHILE TRUE DO END` • `null` • scope = `{num : Integer = NIL}`
`Ast.Stmt.Return`	Validates a return statement. Throws a `RuntimeException` if: • The value is not assignable to the return type of the method it in. ◦ As hinted in `Ast.Method`, you will need to coordinate between these methods to accomplish this. Note: This method will only be called as part of visiting a method, since otherwise there would not be a return type to consider. Returns `null`.	• `RETURN 1;`, return type = `Integer` ◦ success • `RETURN 1;`, return type = `String` ◦ throws `RuntimeException`
`Ast.Expr.Literal`	Validates and sets type of the literal as described below. You will need to make use of `instanceof` to figure out what type the literal value is (remember to distinguish between the type in our language and the type of the Java object!). • `Nil`, `Boolean`, `Character`, `String`: No additional behavior. • `Integer`: Throws a `RuntimeException` if the value is out of range of a Java `int` (32-bit signed int). There are methods in `BigInteger` that can help with this, but make sure to throw a `RuntimeException`! • `Decimal`: Throws a `RuntimeException` if the value is out of range of a Java `double` value (64-bit signed float). This is a bit trickier than the previous one, but the method you should use here is `BigDecimal#doubleValue()`. Check the Javadocs to see what happens if the value doesn't fit into a `double` and go from there. Returns `null`.	• `TRUE` ◦ `ast.getType() == Boolean` • `2147483647` ◦ `ast.getType() == Integer` • `9223372036854775807` ◦ throws `RuntimeException`
`Ast.Expr.Group`	Validates a group expression, setting it's type to be the type of the contained expression. Throws a `RuntimeException` if: • The contained expression is not a binary expression (since this is the only type of expression that is affected by precedence). Returns `null`.	• `(1)` ◦ throws `RuntimeException` • `(1 + 10)` ◦ `ast.getType() == Integer`
`Ast.Expr.Binary`	Validates a binary expression according to the specific operator below, setting it's type to the appropriate result type for the operation. • `AND`/`OR`: ◦ Both operands must be `Boolean`. ◦ Result type will be `Boolean`. • `<`/`<=`/`>`/`>=`/`==`/`!=`: ◦ Both operands must be `Comparable` and of the *same type*. ◦ Result type will be `Boolean`. • `+`: ◦ If either side is a `String`, the result is a `String` (and the other side can be anything). ◦ Otherwise, the left hand side must be an `Integer`/`Decimal` and both the right hand side and result type are the same as the left. • `-`/`*`/`/`: ◦ The left hand side must be an `Integer`/`Decimal` and both the right hand side and result type are the same as the left. Returns `null`.	• `TRUE AND FALSE` ◦ `ast.getType() == Boolean` • `TRUE AND "FALSE"` ◦ throws `RuntimeException` • `"Ben" + 10` ◦ `ast.getType() == String` • `1 + 10` ◦ `ast.getType() == Integer` • `1 + 1.0` ◦ throws `RuntimeException`
`Ast.Expr.Access`	Validates an access expression and sets the variable of the expression (`Ast.Expr.Access#setVariable`), which internally sets the type of the expression to be the type of the variable. The variable is a field of the receiver if present, otherwise it is a variable in the current scope. Returns `null`.	• `variable`, scope = `{variable: Integer}` ◦ `ast.getType() == Integer` • `object.field`, scope = `{object: ObjectType {field: Integer}` ◦ `ast.getType() == Integer`
`Ast.Expr.Function`	Validates a function expression and sets the function of the expression (`Ast.Expr.Function#setFunction`), which internally sets the type of the expression to be the return type of the function. The function is a method of the receiver if present, otherwise it is a function in the current scope. Additionally, checks that the provided arguments are assignable to the corresponding parameter types of the function. • **IMPORTANT**: The first parameter of a method (retrieved from the receiver) is the object it is being called on (like `self` in Python). Therefore, the first argument is at index `1` in the parameters, not `0`, **only** for methods. ◦ This is a bit of a weird quirk, but every design decision has tradeoffs. Returns `null`.	• `function()`, scope = `{function/0: () -> Integer}` ◦ `ast.getType() == Integer` • `object.method()`, scope = `{object: ObjectType {method/1: (Any) -> Integer}}` ◦ `ast.getType() == Integer`

Provided Code

The following files are provided to help you help implement the Analyzer. This includes new versions of some of the provided files, including a updated `ParserTests.java` for handling types.

- Source Files (`src/main/java/plc/project`)
  - Analyzer.java
  - Ast.java
  - Environment.java
  - Scope.java
- Test Files (`src/test/java/plc/project`)
  - AnalyzerTests.java
  - ParserTests.java