Project Checkpoint 4: Analyzer 75 Points Possible

4/14/2025

Add Comment

3/31/2025 to 4/16/2025

Details

In this assignment, you will implement the analyzer for our language. This is the first step in switching from a more dynamically typed, interpreted language to a statically typed, compiled one. The job of the analyzer is to perform semantic analysis over the source code prior to execution, which mostly involves type checking.

Submission

You will submit a Zip file of the entire `src` folder (containing just the single `src` folder), which should contain both your implementation (`src/main/java`) and tests (`src/test/java`). On Windows, right-click on the src folder and "Compress to ZIP file" should have the expected behavior here.

- The first test submission is **Wednesday, April 9**: **Project Checkpoint 4: Analyzer (TEST 1 - Expr) (https://ufl.instructure.com/courses/523541/assignments/6419674)** .
  - This covers all rules, however only `expression` tests (with the exception of `object_expr`) are graded.
    - Note: We will be adjusting the grading scale from previous test submissions to be a bit more lenient around edge cases.
- The second test submission is **Saturday, April 12**: **Project Checkpoint 4: Analyzer (TEST 2 - All) (https://ufl.instructure.com/courses/523541/assignments/6451481)** .
  - This covers all rules, with statements (and `object_expr`) being graded.
- The final submission is **Monday, April 14**.

Project Setup

See the following links for provided code and setup instructions:

- **PlcProject (Analyzer Patch).zip (https://ufl.instructure.com/courses/523541/files/96225796?wrap=1)** **(https://ufl.instructure.com/courses/523541/files/96225796/download?download_frd=1)** : Contains only new `analyzer` files and updates to `Ast.java`/`Main.java`, plus an update to `RuntimeValue.java` for debugging primitive class types. These files should be added to your **existing project** (i.e. Lexer + Parser + Evaluator) following the same folder/package structure

as provided.

- **Project Setup (IntelliJ & Gradle) (https://ufl.instructure.com/courses/523541/pages/project-setup-intellij-and-gradle)**

Analyzer Overview

Recall that the job of semantic analysis is determine if the semantic meaning of the AST is valid according the specification of the language. This can generally be broken down into the following categories:

- Structural AST requirements that *could* have been enforced by the Parser, but make more sense to handle within semantic analysis (typically best when the meaning is clear but the operation itself is invalid to allow IDE to error gracefully).
- Resolution of types, variables, and functions; especially in "larger" programs that may require linking multiple source files together to identify where variables/functions are located so they can be compiled with references to the right locations.
- Type checking, which utilizes static typing to traverse the program and identify unsafe/invalid uses of data throughout the program. This is most helpful in large projects and in languages which have a sophisticated type system, otherwise it's easy to end up "fighting" the compiler to get code that is correct dynamically to compile statically.
  - Our language will have just *slightly* above the bare minimum for types. In fact, there are many functions used in the Evaluator that we won't be able to represent accurate types for - e.g. `log` (which needs generic types to represent that the argument and return value have the same type) and `list` (which functions that support taking a variable number of arguments, aka varargs).

While our analyzer will be limited (especially the type system) the focus here is on how we can move validation from runtime in the evaluator to compile-time in the analyzer, even though we don't actually have runtime values. You should find that the analyzer is quite similar in logic/structure to the evaluator!

Crafting Interpreters

Unfortunately, crafting interpreters doesn't get into the aspects of semantic analysis or compile-time validation. This territory gets much closer into the "each language figures out it's own thing and they're all slightly broken" territory, *especially* with sophisticated type systems.

Grammar/Ast Changes

The grammar and AST of our language has been updated to support types. These types are optional in the AST (thus nothing should break in the Evaluator) however you will need to update your Parser accordingly. These changes will be graded as part of the final submission.

```
let_stmt ::= 'LET' identifier (':' identifier)? ('=' expr)? ';'
def_stmt ::= 'DEF' identifier '(' (identifier (':' identifier)? (',' identifier (':' identifier)?)*)?
')' (':' identifier)? 'DO' stmt* 'END'
```

Specification

The following table lists all AST types and their specification for static analysis. Each `visit(Ast)` method returns the corresponding `Ir` class, i.e. `visit(Ast.Stmt.Let)` returns `Ir.Stmt.Let`.

- Unlike the evaluator, the order semantic analysis is performed is less important since there's no side effects from execution. However:
  - You still need to ensure that the `scope` used for analysis is accurate and has the right variables defined (or not defined!).
  - It's generally best to keep analysis order consistent with evaluation order for clarity and easier review/debugging.

If an error should be thrown, this should use the provided `AnalyzeException` class. This class takes a `String message` - this should be a descriptive message to help identify the issue on your side; tests will not use this message.

AST Type	Description
`Source`	Analyzes a program source, with the following behavior: • Analyze all statements sequentially.

`Stmt.Let`
(in lecture)
	Analyzes a `LET` statement, with the following behavior:
	• Define the variable `name` (which must not already be defined) in the *current scope*.
	  • The `type` is the first of the ast's `type` (if present, which must also be a type in `Environment.TYPES`), the value's type (if present), or else the type `Any`.
	• The `value`, if present, must be a subtype of the variable's type.
	  • Note: This must be analyzed *prior* to defining the variable (as it's needed for type inference and the expression shouldn't have access to the variable).

`Stmt.Def`	Analyzes a `DEF` statement, with the following behavior:
	1. Define the function `name` (which must not already be defined) in the *current scope* with a type of `Type.Function`.
	  • Parameter `names` must be unique.
	  • Parameter `types` and the function's `returns` type must all be in `Environment.TYPES`; if not provided explicitly the type is `Any`.
	2. In a new child scope:

	1. Define variables for all parameters. 2. Define the variable `$RETURNS` (which cannot be used as a variable in our language) to store the return type (see `Stmt.Return`). 3. Analyze all body statements sequentially. Note: We will not be performing control flow analysis to ensure that the function returns a value.
`Stmt.If`	Analyzes an `IF` statement, with the following behavior: 1. Analyze the ast's `condition`, which must be a subtype of `Boolean`. 2. Analyze both the then/else bodies, each within their own new child scope. • Evaluation will only evaluate one, but for purposes of compilation we need to look at both!
`Stmt.For`	Analyzes a `FOR` loop, with the following behavior: 1. Analyze the ast's `expression`, which must be a subtype of `Iterable`. 2. In a new child scope: 1. Define the variable `name` to have type `Integer` (our language will require all `Iterables` to be of `Integers`). 2. Analyze all body statements sequentially.
`Stmt.Return`	Analyzes a `RETURN` statement, with the following behavior: 1. Ensure the variable `$RETURNS` is defined (see `DEF`), which contains the expected return type. • If this variable isn't defined, it means we're returning outside of a function! 2. Verify the type of the return value, which is `Nil` if absent, is a subtype of `$RETURNS`.
`Stmt.Expression`	Analyzes the contained expression with no additional restrictions.
`Stmt.Assignment` (partially in lecture)	Analyzes an assignment statement, with the following behavior: • The receiver must be an `Ast.Expr.Variable` or `Ast.Expr.Property`. • If it is a `Variable`, that variable must be defined. The value must be a subtype of the variable's type. • If it is a `Property`, validate the property's receiver/name in the same was as `Ast.Expr.Property`. The value must be a subtype of the property's type.

`Expr.Literal`	Analyzes a literal expression, with the following behavior: • The expression type corresponds with the type of the value as used by our language (provided).
`Expr.Group`	Analyzes the contained expression with no additional restrictions.
`Expr.Binary`	Analyzes a binary expression, with the following behavior: • `+`: If either operand is a `String`, the result is a `String`. Otherwise, the left operand must be a subtypes of `Integer`/`Decimal` and right must be the same type, which is also the result type. • `-`/`*`/`/`: The left operand must be either an `Integer`/`Decimal` and the right must be the same type, which is also the result type. • `<`/`<=`/`>`/`>=`: The left operand must be a subtype of `Comparable` and the right must be the same type. The result is a `Boolean`. • `==`/`!=`: Both operands must be a subtype of `Equatable`. The result is a `Boolean`. • `AND`/`OR`: Both operands must be a subtype of `Boolean`. The result is a `Boolean`.
`Expr.Variable` (in lecture)	Analyzes a variable expression, with the following behavior: 1. Ensure the variable `name` is defined and resolve it's type.
`Expr.Property`	Analyzes a variable expression, with the following behavior: 1. Analyze the `receiver`, which must be an instanceof `Type.Object`. 2. Ensure that `name` is defined in the object's scope and resolve it's type.
`Expr.Function`	Analyzes a function expression, with the following behavior: 1. Ensure the function `name` is defined and resolve it's type, which must be an instanceof `Type.Function`. 2. Analyze all arguments sequentially, ensuring that each argument is a subtype of the function's corresponding parameter type. 3. The expression type is the function type's return type.
`Expr.Method`	Analyzes a function expression, with the following behavior: 1. Analyze the `receiver`, which must be an instanceof `Type.Object`. 2. Ensure the function `name` is defined in the object's scope and resolve it's type, which must be an instanceof `Type.Function`. 3. Analyze all arguments sequentially, ensuring that each argument is a subtype of the function's corresponding parameter type. • **Important**: Unlike the Evaluator, the types for methods will not have the receiver as a parameter (thus, arguments and parameters should have the same size). See the changelog for details.

	4. The expression type is the function type's return type.
`Expr.ObjectExpr`	Analyzes an object expression, with the following behavior: 1. The name of the object must **not** be a type in `Environment.TYPES`. 2. Analyze all fields, which must have unique names. The analysis follows the same semantics as `LET` statements, however should define the variable in the object's scope. 3. Analyze all methods, which must have unique names (and are unique with fields as well). The analysis follows the same semantics as `DEF` statements, however should also define the variable `this` as an implicit parameter with the type of the object. • **Important: Unlike methods at runtime, the "this" receiver is NOT part of Type.Function parameters.**
`requireSubtype` (partially in lecture)	Throws an `AnalyzeException` if `subtype` is not a subtype of `supertype`. Our language defines subtyping as follows: • All types are subtypes of `Any` (similar to Java's `Object`). • All types are subtypes of themselves (per `.equals()`) and `Any`. • `Nil`, `Comparable` (and all subtypes), `Iterable` are subtypes of `Equatable`. • `Boolean`, `Integer`, `Decimal`, `String` are subtypes of `Comparable`.

Changelog

Date	Change
Apr. 4	• Update the provided files with corrections to `Environment.java` and `AnalyzerTests.java` (see below). ◦ `Environment.java` now has method types for `object` without the receiver parameter. ◦ `AnalyzerTests.java` has corrections for function return tests (missing invocation IR), integer literals (wrong type), and the Hello World program (wrong function name); plus minor changes to the test function for better error messaging. • **Important:** Change the representation of Method types (using `Type.Function`) to *only* have argument types. In other words, the method receiver type (the object's type itself) is no longer the first parameter like in the Evaluator.

	○ TLDR: This leads to recursion issues in equals/toString when printing the object, and in general is a bit awkward to work with in tests. Also simplifies the implementation of ObjectExpr.

Previous (https://ufl.instructure.com/courses/523541/modules/items/11874741)

Next (https://ufl.instructure.com/courses/523541/modules/items/11874744)