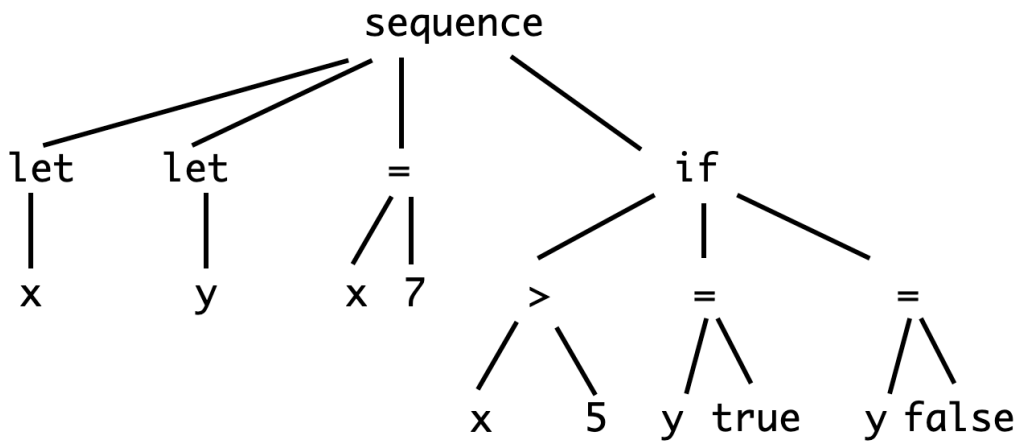
Homework 2: Design Patterns

(out 10/8/2023, due 10/15/2024 @ 11:59pm)

The main purpose of this problem set is to give you an opportunity to work with design patterns. This is an individual assignment, but you may seek help about installing and using the TypeScript toolset, or about the TypeScript language itself. Please refer to the tutorials on the course home page. Your solution may not depend on any third-party libraries other than those provided in the handout archive (you may not npm install any packages to use for this assignment). To get started, please download use the starter code that is bunded with this assignment in Canvas. Note that making changes (e.g. adding methods) to this starter code is allowed.

1. Background: Abstract Syntax Trees

In this homework, you will be working with Abstract Syntax Trees. An *Abstract Syntax Tree (AST)* is a tree representation of a program that reflects the *structure* of that program. ASTs are widely used in compilers, where the construction of an AST is typically the first step in translating source files to an executable binary. ASTs are also widely used in Software Engineering tools for refactoring, clone detection, etc. For example, an AST for the following program: `let x; let y; x = 7; if (x > 5) { y = true } else { y = false }` can be visualized as follows:



For the first part of this homework, a set of classes has been provided to you that model ASTs for a tiny programming language that includes the following features: variable declarations, boolean literals, integer literals, binary "+" and ">" expressions, variable reference expressions, and sequences of statements. These classes are organized as a class hierarchy in which `ASTNode (IASTNode interface)` is the root. AST nodes can be classified as statements (see the `IStatement` interface) or expressions (`IExpression`), where `IStatement` has subclasses

`AssignmentStmt`, `DeclarationStmt`, `IfStmt` and `SequenceStmt` and where `IExpression` has subclasses `BooleanLiteral`, `IntegerLiteral`, `AddExpression`, `GreaterThanExpression`, and `ReferenceExpression`. `IASTNode` declares a method `text()` that generates a textual representation for the subtree rooted at that AST node. A small test suite is provided that checks that the `text()` method produces the expected results. Please install the code and confirm that the tests pass.

1. Counting Nodes in an AST (50 points)

For this task, please use the provided `IASTNode` hierarchy as a starting point. Please extend the hierarchy so that it can accept an `IASTVisitor` by adding a new method `accept(visitor: IASTVisitor)` to `IASTNode`, and then implementing this method in the class hierarchy to apply the Visitor design pattern. (Note: Interfaces are present in 'src/interfaces/' and class implementations in 'src/ast-nodes/')

Then, implement a class `CountVisitor` (located at 'src/CountVisitor.ts' which implements `IASTVisitor`) that counts, for each variable, how many times it is referenced. You may assume that the program is correct (i.e., all variables are declared, and no variable is declared more than once). Provide `CountVisitor` with methods for retrieving the count for a variable after the visit has been completed. Below is an example of a test that should pass.

```
    it('count references in small AST', (): void => {

    const declx: IStatement = new DeclarationStmt('x');
    const decly: IStatement = new DeclarationStmt('y');
    const declz: IStatement = new DeclarationStmt('z');
    const x: ReferenceExpression = new ReferenceExpression('x');
    const y: ReferenceExpression = new ReferenceExpression('y');
    const z: ReferenceExpression = new ReferenceExpression('z');
    const one: ILiteralExpression = new IntegerLiteral(1);
    const two: ILiteralExpression = new IntegerLiteral(2);
    const assign1tox: IStatement = new AssignmentStmt(x, one);
    const assign2toy: IStatement = new AssignmentStmt(y, two);
    const comparison: IExpression = new GreaterThanExpression(x, y);
    const assignxtoz: IStatement = new AssignmentStmt(z, x);
    const assignytoz: IStatement = new AssignmentStmt(z, y);
    const ifstmt: IStatement = new IfStmt(comparison, assignxtoz, assignytoz);
    const incrementz: IStatement = new AssignmentStmt(z, new AddExpression(z, one));
    const seq: IStatement = new SequenceStmt([declx, decly, declz, assign1tox,
assign2toy, ifstmt, incrementz]);

    const countVisitor: CountVisitor = new CountVisitor();

    seq.accept(countVisitor);

    expect(countVisitor.getCount('x')).toEqual(3);
```

```
    expect(countVisitor.getCount('y')).toEqual(3);
    expect(countVisitor.getCount('z')).toEqual(4);
  });
```

Please provide a set of tests in a file '`CountVisitor.spec.ts`' that includes this test, as well as several additional tests.

Grading breakdown for this task:

- 10 points: Automated GradeScope tests of your `CountVisitor` (5 tests, 2 points each)
- 20 points: Manual grading of your `CountVisitor` (and any modifications to the AST classes and interfaces that you may make), checking primarily to ensure that you applied the Visitor pattern correctly, have reasonable variable names, and that the behavior is overall correct (beyond the automated tests that we provide).
- 20 points: Manual grading of your tests. Construct test cases to check the behavior of your `CountVisitor` in the presence of different AST structures.

2. Finding Type Errors in an AST (50 points)

For the next task, please create a type-checker [1] for your AST that detects the following types of problems:

- Type errors in an `AddExpression` where one or both of the operands is a `BooleanLiteral`
- Type errors in a `GreaterThanExpression` where one or both of the operands is a `BooleanLiteral`
- Duplicate variable declarations (i.e., situations where your AST contains multiple `Declarations` that refer to the same variable)
- Missing variable declarations (i.e., situations where your AST contains a `ReferenceExpression` that refers to a variable for which no Declaration exists)

Note: you do not need to handle the case where one or both of the operands of a binary expression is a variable.

To represent these errors, please define an interface `ITypeError` (located at 'src/interfaces/ITypeError.ts') that has a method `getMessage` that produces a human-readable representation of the error message. Then, define subclasses (in 'src/errors/') `DuplicateDeclarationError`, `MissingDeclarationError`, `BadOperandInAddError`, `BadOperandInComparisonError` and `BadConditionInIfError` to represent the errors mentioned above.

Your errors should be formatted as follows:

- `DuplicateDeclarationError.message()` should return the exact string `Duplicate declaration of variable: {varName}`, where `{varName}` is the name of the variable that has been defined multiple times.
- `MissingDeclarationError.toString()` should return the exact string `Missing declaration for variable: {varName}`, where `{varName}` is the name of the missing variable.
- `BadOperandInAddError.message()` should return the exact string `Bad operand in add expression: {expr}`, where `{expr}` is the string returned by calling `text()` on the offending expression.
- `BadOperandInComparisonError.message()` should return the exact string `Bad operand in comparison expression: {expr}`, where `{expr}` is the string returned by calling `text()` on the offending expression.
- `BadConditionInIfError.message()` should return the exact string `Bad condition in if statement: {cond}`, where `{cond}` is the string returned by calling `text()` on the offending condition.
-

Your solution must implement the type checker as a subtype `TypeCheckVisitor` of `IASTVisitor`. Your `TypeCheckVisitor` must define the method public `getErrors() : Array<ITypeError>`, which returns an array of all of the errors accumulated by the visitor. Again, you should feel free to modify any of the supplied AST classes and interfaces, and/or define new classes/interfaces that you feel would be appropriate.

Grading breakdown for this task:

- 10 points: Automated GradeScope tests of your `TypeCheckVisitor` (5 tests, 2 points each)
- 20 points: Manual grading of your `TypeCheckVisitor` (and any modifications to the AST classes and interfaces that you may make), checking primarily to ensure that you applied the Visitor pattern correctly, have reasonable variable names, and that the behavior is overall correct (beyond the automated tests that we provide).
- 20 points: Manual grading of your tests. Construct test cases to check the behavior of your `TypeCheckVisitor` in the presence of different AST structures.

[1] A type-checker is a component of an interpreter or compiler for checking static safety properties.

Submission:

- Run `npm run zip` to generate `submission.zip` and upload to Gradescope.

Please be reminded of the course late submission policy: assignments will be accepted up to 24 hours past the deadline at a penalty of 25%, and not accepted beyond 24 hours past the deadline.