CSC-223
Assignment 6
Work in a Pair

AST Visitors

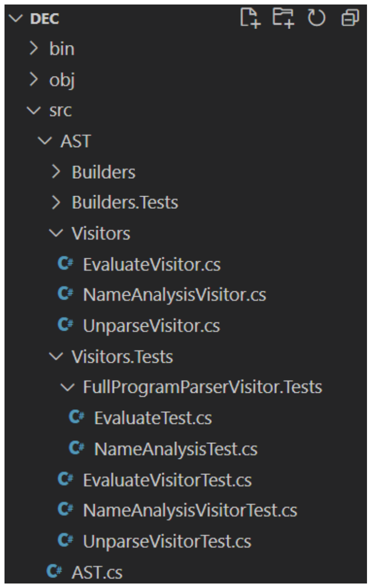
In the previous assignment, we implemented a language parser for DEC that constructs Abstract Syntax Trees (ASTs). For this assignment, we will focus on implementing the Visitor design pattern to encapsulate several operations over the AST. This will help us maintain a separation of concerns and avoid cluttering our AST classes with traversal-specific code.

What You Need to Do: Continuing From the Last Assignment

You may continue to work with your partner via the same repository or a new repository for this assignment; the same repository is recommended. Please see the "Git and Github Collaboration" section in the first assignment for details.

What You Need to Do: Adding to the Project Structure

Visitors will be defined in the AST folder (adjacent to the Builders).



```
DEC
  bin
  obj
  src
    AST
      Builders
      Builders.Tests
      Visitors
        EvaluateVisitor.cs
        NameAnalysisVisitor.cs
        UnparseVisitor.cs
      Visitors.Tests
        FullProgramParserVisitor.Tests
          EvaluateTest.cs
          NameAnalysisTest.cs
        EvaluateVisitorTest.cs
        NameAnalysisVisitorTest.cs
        UnparseVisitorTest.cs
      AST.cs
```

Figure 1: The desired project structure including AST Visitors and tests.

What We Did: Adding Methods to AST Nodes

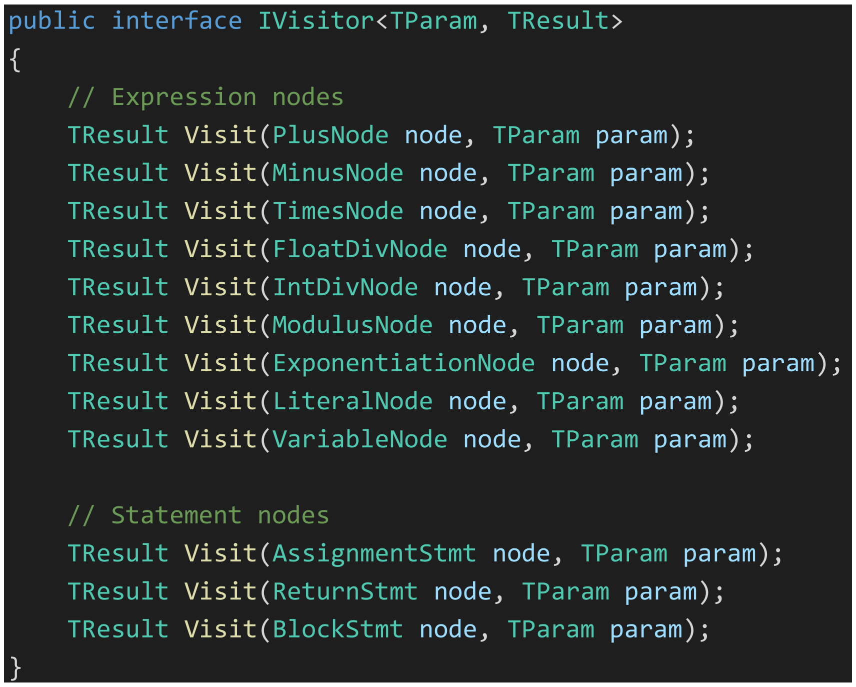
In earlier assignments, we implemented traversal techniques (like unparsing) by adding methods directly to AST classes. While this approach worked, it has several drawbacks. Each new operation requires adding methods to each node class, causing AST classes to become cluttered with operation-specific code. Related code for a single operation is possibly spread across multiple files, making maintenance difficult as the number of operations grows. Our goal is to address these issues by implementing the Visitor design pattern over the AST, which helps separate functionality from data structures and keeps related code in dedicated files.

Background and Implementation Information: The Visitor Design Pattern

The visitor pattern promotes better code organization. All logic related to a single operation (like name analysis, unparsing, or evaluation) is centralized in one visitor class rather than being distributed across multiple node classes. This makes the code easier to understand, maintain, and test. When debugging, you know exactly where to look for issues related to a specific operation. Additionally, visitors can maintain state across the traversal process, making it easier to implement operations that require context or accumulate information as they process the AST.

The visitor design pattern exploits a two-way dispatching mechanism (often called double dispatch) that routes method calls based on both the type of the visitor and the type of the node being visited. This approach provides tremendous flexibility—we can add new operations by creating new visitor classes without modifying any of the existing AST node classes. Similarly, if we need to add new node types, we can update the visitor interface to accommodate them without changing existing visitor implementations.

Interface Implementation and Visit. The code begins by defining an interface, `IVisitor` as shown in Figure 2, to facilitate what it means for a visitor to work on AST classes.



```
public interface IVisitor<TParam, TResult>
{
    // Expression nodes
    TResult Visit(PlusNode node, TParam param);
    TResult Visit(MinusNode node, TParam param);
    TResult Visit(TimesNode node, TParam param);
    TResult Visit(FloatDivNode node, TParam param);
    TResult Visit(IntDivNode node, TParam param);
    TResult Visit(ModulusNode node, TParam param);
    TResult Visit(ExponentiationNode node, TParam param);
    TResult Visit(LiteralNode node, TParam param);
    TResult Visit(VariableNode node, TParam param);

    // Statement nodes
    TResult Visit(AssignmentStmt node, TParam param);
    TResult Visit(ReturnStmt node, TParam param);
    TResult Visit(BlockStmt node, TParam param);
}
```

Figure 2: The `IVisitor` interface to facilitate visitor implementations over the DEC AST classes.

The `IVisitor` interface defines a contract for all visitor classes and serves as a central hub for visitor operations. It declares a `Visit` method for each concrete AST node type that might be encountered during traversal. For the DEC language AST, this means separate methods for expressions (e.g., plus, minus, literals, etc.) and all statements (assignments, returns, blocks).

Using generic parameters(`TParam`, `TResult`) with visitors provides flexibility that makes the pattern more powerful and reusable: enabling customized context data transfer and flexible return types without changing the visitor structure. `TParam` allows different visitors to accept different input data (like symbol tables or indentation levels), while `TResult` enables visitors to return different types of output (like booleans for analysis or strings for unparsing) without changing the visitor structure. For example, `UnparseVisitor` might use `int` (for the level of indentation) and `string` for output, while our `NameAnalysisVisitor` might use `SymbolTable<string, object>` as its parameter type and `bool` as its result type.

Accept. The visitor pattern requires close cooperation between the visitor and the visited classes in the AST. While the visitor interface provides half of the double-dispatching mechanism, the `Accept` method in AST nodes provides the crucial other half. Shown below is the abstract method declaration needed in all abstract AST classes like the `ExpressionNode` class.

```
public abstract class ExpressionNode
{
    public abstract TResult
    Accept<TParam, TResult>(IVisitor<TParam, TResult> visitor, TParam param);
}
```

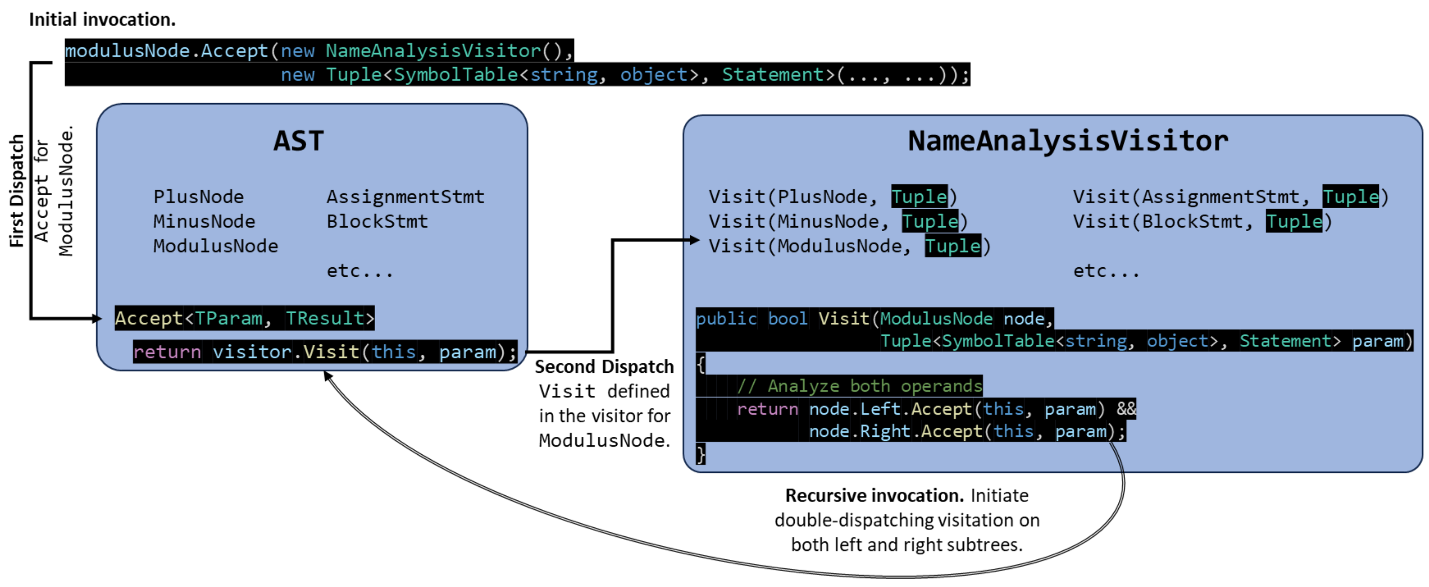
Each concrete AST node class will then provide an implementation of this method, typically with a one-line implementation:

```
public override TResult
Accept<TParam, TResult>(IVisitor<TParam, TResult> visitor, TParam param)
{
    return visitor.Visit(this, param);
}
```

Double-Dispatching. The implementation of `Accept` methods invoking a `Visit` method is what enables double-dispatching. When a client (i.e., a method in a visitor like `UnparseVisitor`) calls `node.Accept(visitor, param)`, two dispatches occur:

1. The first dispatch selects the appropriate `Accept` method based on the type of node at runtime.
2. The second dispatch happens inside `Accept` when it calls `visitor.Visit(this, param)`, which selects the appropriate `Visit` method based on the static type of `this`.

A visual example of the double dispatching process is given in Figure 3.



```
modulusNode.Accept(new NameAnalysisVisitor(),
                   new Tuple<SymbolTable<string, object>, Statement>(..., ...));
```

```
Accept<TParam, TResult>
  return visitor.Visit(this, param);
```

```
public bool Visit(ModulusNode node,
                  Tuple<SymbolTable<string, object>, Statement> param)
{
    // Analyze both operands
    return node.Left.Accept(this, param) &&
           node.Right.Accept(this, param);
}
```

Figure 3: A visual representation of double-dispatching with a `NameAnalysisVisitor` object. A node of type `ModulusNode` initiates visitation (first dispatch) calling `Accept` which subsequently calls the `Visit` (second dispatch) defined in the visitor class.

Double-dispatching solves the problem of selecting different operations based on the concrete types of both the visitor and the node. Without it, we would need complex type-checking or casting logic to determine which operation to perform on which node type.

Another double-dispatching example. To understand the power of this approach, consider what happens when the code shown below executes.

```
ExpressionNode expr = new PlusNode(new LiteralNode(5), new VariableNode("x"));
bool result = expr.Accept(nameAnalysisVisitor, symbolTable);
```

Even though `expr` is declared as an `ExpressionNode`, the correct `Accept` method for `PlusNode` is called at runtime. Inside that method, `this` has the static type `PlusNode`, so the compiler selects `visitor.Visit(PlusNode, TParam)`. This happens without any explicit type checking or casting. This elegant mechanism lets us separate traversal logic from node structure while maintaining type safety and avoiding the performance overhead of runtime type identification.

Come on Alvin! Unacceptable! The visitor pattern with double dispatch is inherently complex, and simply reading about these concepts will leave you questioning life itself. However, the elegance of this design pattern only becomes apparent when you implement it yourself and see how the pieces work together. Do not expect to fully grasp the mechanics from this explanation alone. You need to write the code, debug the interactions, and experience the "aha!" moments that come from seeing double dispatch resolve method calls at runtime. Dive in and build it; the power of the pattern will reveal itself through practice.

What You Need to Do: Implement Visitors

The first step is to add the definition of `IVisitor` from Figure 2 to the `AST.cs` file (within the AST namespace, but not in any of the AST classes). We then need to add `Accept` methods to all abstract and concrete classes in the AST. (For simplicity and consistency with old tests, keep the original `Unparse` methods we wrote in the AST classes.)

We will then implement and test three visitors. Each serves a distinct purpose in our DEC processing framework.

1. **`UnparseVisitor`** reconstructs the original source code from the AST, maintaining proper indentation and formatting while traversing the tree, essentially performing the inverse operation of parsing.
2. **`EvaluateVisitor`** 'executes' the program represented by the AST, calculating expression values, handling variable assignments, respecting scope rules, and returning the final computation result, effectively acting as an interpreter for our language.
3. **`NameAnalysisVisitor`** traverses the AST to verify variables are properly defined before being used, building symbol tables as it progresses through different scopes and recording errors for undefined variables while maintaining context about where errors occur.

`UnparseVisitor`. The `UnparseVisitor` reconstructs the original source code from the AST, essentially performing the reverse operation of parsing. As it traverses the tree, it builds a string representation of the program, carefully managing proper indentation and formatting to ensure the output is readable and correctly structured. For expressions, it ensures that operator precedence is clearly represented through appropriate parenthesization. For statements, it handles proper formatting of assignments, return statements, and blocks with their nested scopes. (Note: unparsing parentheses around singleton tokens is optional therefore, this will not be a true inverse operation.)

This visitor is particularly valuable for debugging purposes, allowing developers to verify that the AST correctly represents the intended program structure. It also serves as a pretty-printer for the language, potentially reformatting code in a standardized way. Like a previous assignment, the `UnparseVisitor` will use an integer parameter to track indentation depth and returns string values that build up the complete program text.

```
public class UnparseVisitor : IVisitor<int, string>
```

`EvaluateVisitor.` The `EvaluateVisitor` acts as an interpreter for our language, executing the program represented by the AST to produce actual results. It handles arithmetic operations, variable assignments, and control flow as it traverses the tree. For expressions like `PlusNode` or `ModulusNode`, it recursively evaluates the operands and applies the corresponding operation. For statements like assignments, it updates variable values in the appropriate symbol table. For blocks, it manages scope entry and exit, ensuring variables are properly shadowed.

This visitor must handle runtime concerns such as division by zero, type conversions between integers and floating-point values, and the immediate termination of execution when a return statement is encountered: when a return is encountered, program execution creases and a value is returned. When no return is evident in the code, return the value of the last statement in the block statement that was executed.

The visitor will use a symbol table as its parameter and returns object values that can represent either an integer or a floating-point results (hence, `TResult` is `object` and not explicitly `int` or `double`). This clear separation between syntax (the AST) and semantics (the evaluation) is a key strength of the visitor pattern in language processors.

```
public class EvaluateVisitor : IVisitor<SymbolTable<string, object>, object>
```

By implementing these three visitors, we create a flexible language processing framework that cleanly separates different concerns: syntactic validation (name analysis), representation (unparsing), and execution (evaluation). This modular approach makes the system easier to extend, maintain, and debug.

`NameAnalysisVisitor.` The `NameAnalysisVisitor` performs static analysis on the AST to ensure proper variable usage throughout the program. It uses a specialized tuple parameter `Tuple<SymbolTable<string, object>, Statement>` that serves two crucial purposes: the symbol table tracks variable definitions across scopes, while the `Statement` component provides context about which statement contains the current node being analyzed.

For example, when analyzing a variable reference like `y` in the statement `x := (4 + y)`, the visitor passes the entire assignment statement as the Statement component of the tuple. This contextual information enables more precise error reporting—when an undefined variable is detected, the `Visit` method for `VariableNode` returns `false` and records an error message that identifies exactly which statement contains the problematic reference. As the visitor traverses from statements to their component expressions, it maintains this contextual link, ensuring that even deeply nested expressions can be traced back to their containing statement. This boolean return value propagates up the visitor chain, allowing parent nodes to know if their children encountered errors without stopping the analysis process.

Overall, this visitor systematically builds symbol tables as it processes the AST, respecting scope boundaries created by block statements. It continues analyzing the entire tree even after encountering errors or return statements, ensuring a comprehensive analysis that catches all naming issues in a single pass. By separating name analysis from the AST structure and using the tuple parameter to maintain both scope and statement context, the language processor becomes more modular and provides richer debugging information for developers.

```
public class NameAnalysisVisitor :
    IVisitor<Tuple<SymbolTable<string, object>, Statement>, bool>
```

What You Need to Do: Testing

Your tests fall into two categories that verify different aspects of the visitor implementation. The first category tests each visitor directly by manually constructing AST nodes, creating the necessary parameters (such as symbol tables, indentation levels, or tuples), and then calling the `Accept` method on those nodes with the appropriate visitor and parameters before asserting the expected results. This will be similar to prior labs.

The second category for testing (to be defined in `EvaluateTest` and `NameAnalysisTest`), performs integration testing by writing a complete DEC program as a string, parsing it using `Parser.Parser.Parse(program)` to obtain a `BlockStmt` AST, then invoking the visitor's main method (like `_evaluator.Evaluate(ast)`) on the parsed tree, and finally asserting the results or checking for expected exceptions. The key distinction is that direct visitor tests verify individual visitor methods in isolation, while integration tests verify the complete pipeline from source code through parsing to visitor execution. A sample is given below.

```
[Fact]
public void TestIntDivisionByZero()
{
   // Test integer division by zero exception
   string program = @"{
        return (10 // 0)
    }";

    BlockStmt ast = Parser.Parser.Parse(program);

    var exception = Assert.Throws<EvaluationException>(() =>
                                                      _evaluator.Evaluate(ast));
    Assert.Equal("Division by zero", exception.Message);
}
```

Note that the `@` prefix before a string literal creates a verbatim string in C#, which treats backslashes as literal characters and allows multi-line strings without escape sequences, making it ideal for writing readable test programs.

What follows are standard instructions for generating unit test code.

All tests should be xUnit tests and adhere to the norms of `xUnit` including using `[Fact]` and `[Theory]`.

New to CSC-223! LLMs can be helpful tools for generating testing code for your assignments. When using an LLM to create tests, include prompts like "write thorough tests for all methods", "make sure to test with duplicate data as well" to ensure comprehensive coverage, and "use `[Theory]` and `InlineData` to avoid redundancy in testing code". ***However, be warned that generated code isn't always perfect—your professor has encountered several cases where tests were written incorrectly for the corresponding code.*** Always review each test carefully to verify it is testing what you intend and functioning properly with your implementation. It is also fair to ask the LLM to see if any cases have been missed in the tests that it generates. In fact, it is advised that you ask an LLM to 'check its work'; do not accept a first answer.

Even though you may use an LLM for testing, you will explicitly be asked about test coverage.

What You Need to Do: Commenting

Comment code consistent with the standards stated in the first assignment: when needed, use an LLM to generate comments then refine as needed.

Submitting: Source Code

You will demonstrate your source code to the instructor, in person. Be ready to demonstrate successful execution of all unit tests and describe all tests and code you were required to implement.