CSC-223
Assignment 7
Work in a Pair

Control Flow Graph Generation

In the previous assignment, we implemented the Visitor design pattern to traverse the AST and perform several operations. For this assignment, we will implement a Control Flow Graph (CFG) to analyze a program's execution paths. This analysis is fundamental for program optimization and other compile-time analyses.

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

As shown in Figure 1, our analysis code (and tests) will be scattered throughout the project. `Digraph` is a general directed graph implementation while `CFG` is an inheriting structure from `DiGraph`. Last, code to construct a CFG using another visitor over the AST (`ControlFlowGraphGeneratorVisitor`).

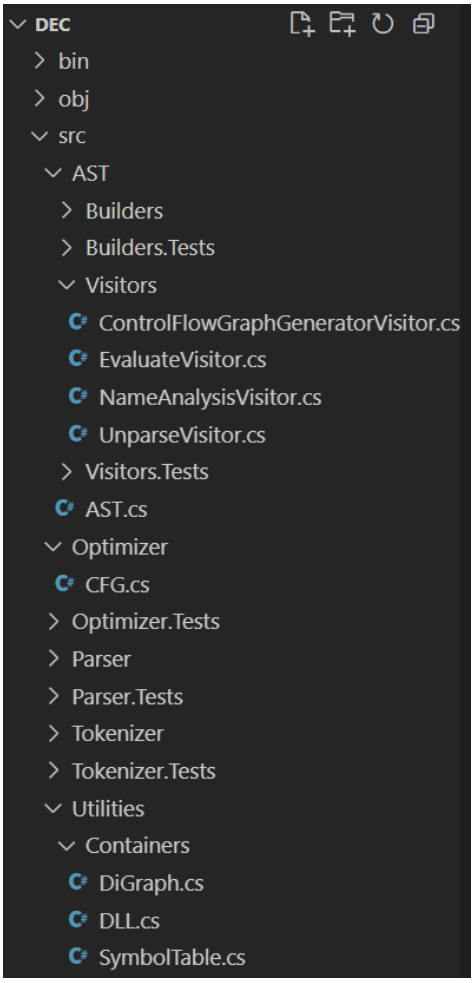

Figure 1: The desired project structure including CFG-related code.

Background: Control Flow Graphs

A Control Flow Graph (CFG) is a directed graph that represents all paths that might be traversed through a program during its execution. In a CFG, **nodes (vertices)** represent program statements or basic blocks (sequences of statements with no branches), while **edges** represent possible transfers of control flow between statements or blocks. The **entry node** represents the starting point of program execution, and **exit node(s)** represent points where program execution can terminate. Together, these elements create a complete map of how control flows through a program.

Indeed, CFGs are essential tools in compiler design, program analysis, and software optimization. They allow us to reason about a program as a behavioral abstraction without considering all possible runtime values. Our language is quite simple so our CFGs will correspondingly be simple. Consider this DEC program and its corresponding CFG in Figure **2**.

```
x := (5)
y := (10)                    [x := 5] → [y := 10] → [z := x + y] → [return z]
z := (x + y)
return (z)
```

Figure 2: An example DEC program and the CFG as a linearization.

With CFGs, each statement is a node; edges represent the flow of control from one statement to the next.

Constant Propagation. As an example of how a CFG may be used, we consider the idea of constant propagation: an optimization that replaces variables with their known constant values *at compile time*. This results in several benefits:

- ***Code size reduction.*** By replacing variables with their constant values, unnecessary variable declarations, assignments, and memory references can be eliminated, resulting in smaller executable code.
- ***Execution speed improvements.*** When constants are propagated, the compiler can perform calculations at compile-time rather than runtime, reducing the computational workload during program execution.
- ***Enabling further optimizations.*** Constant propagation often reveals opportunities for additional optimizations like dead code elimination, strength reduction (replacing expensive operations with cheaper ones), and loop optimization.
- ***Better register allocation.*** With fewer variables needed, register allocation becomes more efficient, potentially reducing memory access operations.

Consider the DEC program and its corresponding equivalent code and constant propagation in Figure 3.

```
x := 5;
y := 10;
z := x + y;                  return 30;
w := z * 2;
return w;
```

Figure 3: An example DEC program before and after constant propagation.

With Figure 3, we observe that `x` is always `5` and `y` is always `10`. We can therefore replace intermediate expressions with their constant values (and evaluate). Thus, `z` is always `15` and it follows `w` is always `30`.

Indeed, CFGs can be a powerful tool toward better code.

What You Need to Do: `DiGraph<T>`

The `DiGraph<T>` class is a generic directed graph implementation that serves as the foundation for the Control Flow Graph. The methods we will implement for the class are described in Table 1. The implementation should use an adjacency list representation involving the `DLL` class you implemented many moons ago. Observe that type `T` is not allowed to be `null` and that the adjacency list is protected.

```
public class DiGraph<T> where T : notnull {
    protected Dictionary<T, DLL<T>> _adjacencyList;
```

Table 1: Methods to be implemented in the `DiGraph<T>` class.

Return Type	Method	Arguments	Description	Exceptions
`bool`	`AddVertex`	`T vertex`	Adds a vertex to the graph if it does not already exist.	
`bool`	`AddEdge`	`T source, T destination`	Adds a directed edge from source to destination, if it does not already exist.	`ArgumentException` when either vertex does not exist in the graph
`bool`	`RemoveVertex`	`T vertex`	Removes a vertex and all edges connected to it.	
`bool`	`RemoveEdge`	`T source, T destination`	Removes a directed edge from source to destination.	`ArgumentException` when either vertex does not exist in the graph
`bool`	`HasEdge`	`T source, T destination`	Checks if an edge exists from source to destination.	
`List<T>`	`GetNeighbors`	`T vertex`	Returns all vertices adjacent to the specified vertex.	`ArgumentException` when the vertex does not exist in the graph
`IEnumerable<T>`	`GetVertices`		Returns all vertices in the graph as an iterable container.	
`int`	`VertexCount`		Returns the number of vertices in the graph.	
`int`	`EdgeCount`		Returns the number of edges in the graph.	
`string`	`ToString`		Returns a string representation of the graph.	

What You Need to Do: CFG

Our implementation will be consistent with the descriptions above: a CFG is a directed graph of statements. The definition should be placed in the `Optimizer` namespace.

```
public class CFG : DiGraph<Statement>
```

The only distinction our CFG will have in this assignment compared to the `DiGraph` is a nullable starting `Statement`. That is, to facilitate analysis of our control flow graphs, we need to store the starting statement of a DEC program as a property [`public Statement? Start { get; set; }`].

What You Need To Do: `ControlFlowGraphGeneratorVisitor`

The `ControlFlowGraphGeneratorVisitor` traverses the AST to build a CFG, implementing the Visitor pattern we established in the previous assignment. This visitor serves as the bridge between the syntactic structure of the program (AST) and its operational execution flow (CFG), translating one representation into the other through a systematic traversal process.

The visitor maintains a CFG and tracks the flow of control through the program, ensuring statements are connected in the order they would execute at runtime. It is advised that, while the AST is traversed, we pass and return `Statement` objects as `TParam` and `TResult` of the `IVisitor` interface. Thus, we can create a linked chain that naturally connects sequential statements, accurately modeling how code would execute at runtime.

Observe that our CFG should only consist of `AssignmentStmt` and `ReturnStmt` objects because `BlockStmt` objects are syntactic units that 'become folded' into the program execution structure represented by the CFG.

What You Need to Do: Testing

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.