Execution Flow Generator Design

Created by Radzivilo(Contractor),Viktor, last modified on Jan 09, 2024

1. Context

In the process of developing a OML compiler, it’s crucial to ensure accurate translation of intermediate code, represented by the Abstract Syntax Tree (AST), into CLR code within .NET assembly. To address this need, the Execution Flow Generator is employed as a key subsystem of the compiler.

The Execution Flow Generator takes the AST as input and generates corresponding CLR code. It leverages Symbol Tables, which encapsulate scopes for the resolution of types, variables, and procedure names. This mechanism ensures that the semantics of the original OML code are preserved in the generated CLR code, facilitating efficient execution.

2. Terms and Definitions

- **Abstract Syntax Tree (AST)**: A data structure used in computer science to represent the structure of a program or code snippet. It is a tree representation of the abstract syntactic structure of source code.
- **CLR Code**: Code that runs under the Common Language Runtime (CLR), a component of Microsoft’s .NET framework that manages the execution of .NET programs.
- **.NET Assembly**: A single deployment unit in .NET that contains a collection of types and resources. It is used as a building block in a .NET application.
- **Execution Flow Generator**: A key subsystem of the compiler that takes the AST as input and generates corresponding CLR code.
- **Symbol Tables**: Data structures used by compilers to store information about the occurrence of various entities such as variable names, function names, objects, classes, interfaces, etc.
- **AST Visitor**: In this document, the AST Visitor refers to a set of classes that share the same interface and each specializes in traversing specific parts of the Abstract Syntax Tree (AST). This design allows for better code modularity and a more effective implementation of the separation of concerns principle. Each class in this set is responsible for traversing a specific part of the AST and performing operations on it. This approach enhances the maintainability and readability of the code.

3. Requirements

3.1. Input Acceptance

The generator should accept an Abstract Syntax Tree (AST) as input, which represents the intermediate code derived from the source OML code.

3.2. Symbol Table Integration

The generator should integrate with Symbol Tables for resolving types, variables, and procedure names, ensuring that identifiers in the source code are correctly interpreted.

3.3. CLR Code Generation

The generator should translate the AST into Common Language Runtime (CLR) code within a .NET assembly, preserving the semantics of the original QuickBasic code.

3.4. Error Handling

The generator should have robust error handling mechanisms, providing meaningful error messages to aid in debugging when an error is encountered during the translation process.

3.5. Performance

The generator should be optimized for performance, ensuring that the compiler can handle large QuickBasic programs efficiently.

3.6. Modularity and Maintainability

The generator should be designed as a modular component of the compiler, allowing for easier maintenance and potential upgrades or modifications in the future.

3.7. Documentation

The generator should be well-documented, including comments in the code and external documentation describing its design, functionality, and usage.

3.8. Readable Output

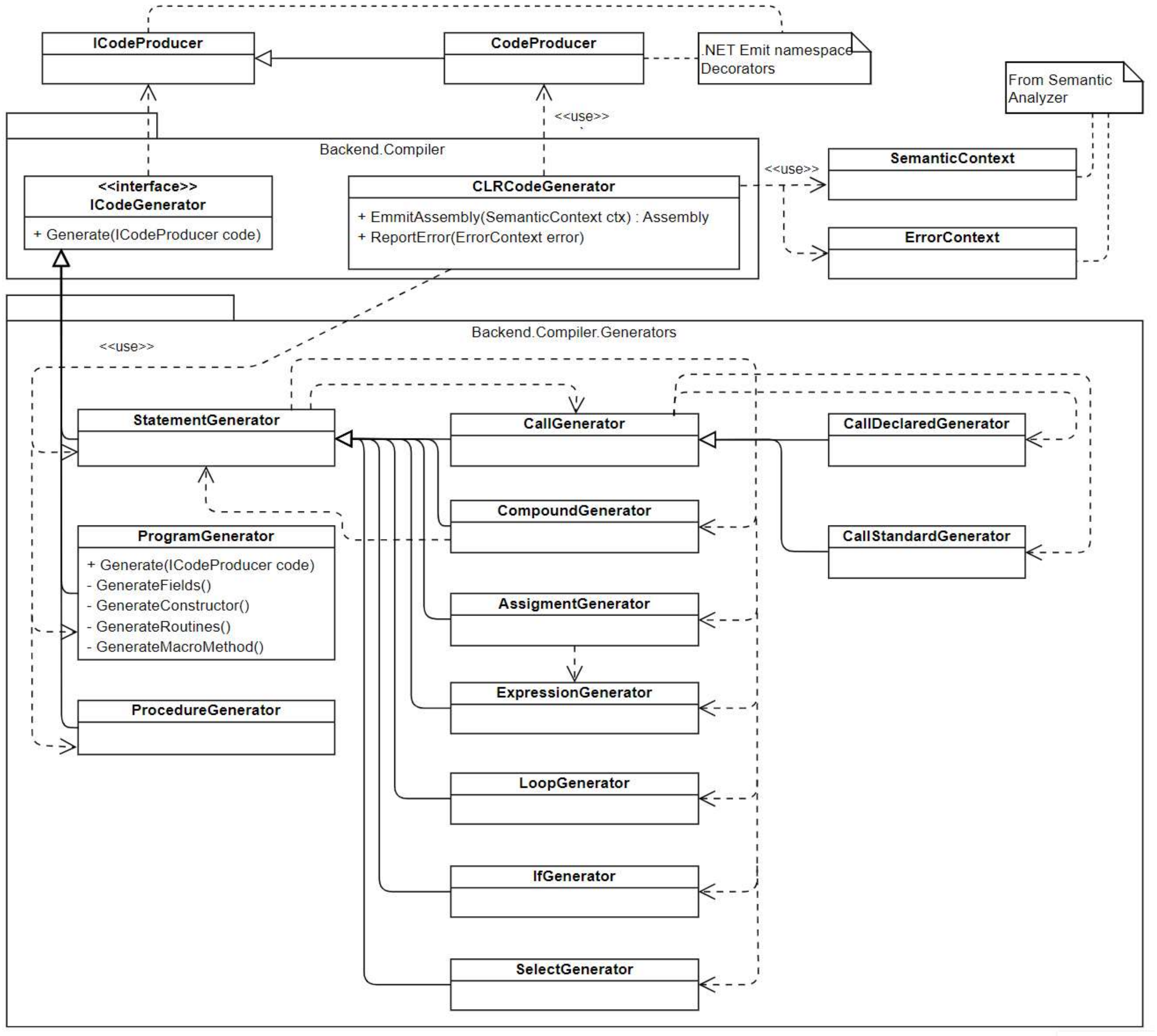
The Execution Flow Generator should present the generated .NET assembly code in a format that is readable by humans, improving the overall transparency of the translation process and providing means for integration testing.

4. Code Generator Design

The `CLRCodeGenerator` is a pivotal class in the design, tasked with emitting .NET assembly based on the provided `SemanticContext`. It employs an AST visitor, implemented as a class hierarchy within the `Backend.Compiler.Generators` namespace. Each class in this hierarchy is responsible for traversing a specific part of the AST and generating corresponding CLR code. There is a direct correlation between the classes of `Backend.Compiler.Generators` and the AST nodes defined in the AST Builder Design document.

Every class that implements the `ICodeGenerator` interface is required to implement the `Generate(ICodeProducer code)` method, which accepts a `ICodeProducer` instance as a parameter and utilizes it for CLR code generation. Additional logic can be implemented as private methods within the class, as demonstrated on the class diagram in the `ProgramGenerator` class.

The `Backend.Compiler.Generators` namespace is designed to be extensible, allowing for the addition of new classes as needed.



Here is detailed breakdown of the class diagram.

4.1. ICodeGenerator

This interface provides a method signature for generating CLR code.

- **Generate -** This method is part of the `ICodeGenerator` interface. It’s designed to generate CLR code using a `CodeProducer` object, which provides the context for generating a .NET module code. The specifics of the code generation process are defined in the classes that implement this interface. The method modifies the `CodeProducer` object, adding the generated CLR code to it. This method does not return any value.

4.2. CodeProducer

To satisfy 2.8 requirement this class is responsible to produce CLR code in form of .NET assembly and in form of human readable text. On the diagram the `CodeProducer` class and other classes in Backend.Compiler.Code namespace acts are decorators for classes from .NET Emit namespace. Please see section 4 for details.

4.3. CLRCodeGenerator

This class emits assembly, generates code, and reports errors.

- **EmmitAssembly -** This method is responsible for generating a .NET assembly from the Abstract Syntax Tree (AST) contained in the provided SemanticContext. It traverses the AST and generates corresponding Common Language Runtime (CLR) code for each node. The Symbol Tables within the `SemanticContext` are used for resolving types, variables, and procedure names during this process. If an error occurs during the code generation process, it is caught and reported using the `ReportError` method.
- **ReportError -** This method is used for handling errors that occur during the code generation process. It takes an `ErrorContext` object as input, which contains information about the error. The method formats a meaningful error message based on the information in the `ErrorContext` and throws an exception with this message. This allows errors to be caught and handled at a higher level in the compiler, aiding in debugging. This method does not return any value.

4.4. SemanticContext

This class provides context for semantic analysis during code generation. See Semantic Analyzer Design document.

4.5. ProgramGenerator

This class generates program parts of the CLR code. The private methods presented on class diagram are just an example and can be missing or have another signature in the final implementation.

4.6. ProcedureGenerator

This class generates procedure (subroutine or function) parts of the CLR code.

4.7. StatementGenerator

This class generates statement parts of the CLR code.

4.8. CallGenerator

This class generates call statements.

4.9. CompoundGenerator

This class generates compound statements, typically blocks of code. The OML does not use explicit code blocks however multiline statements (like `IF` statement for example) used implicit code blocks between `IF` , `ELSE` and `END IF`.

4.10. AssignmentGenerator

This class generates assignment statements, assigning values to variables.

4.11. ExpressionGenerator

This class generates expression statements, representing computations or string operations.

4.12. LoopGenerator

This class generates loop statements, representing iterative blocks of code.

4.13. IfGenerator

This class generates if statements, representing conditional blocks of code.

4.14. SelectGenerator

This class generates select statements, representing multi-way branches in the code.

4.15. CallDeclaredGenerator

This class generates call statements for declared procedures.

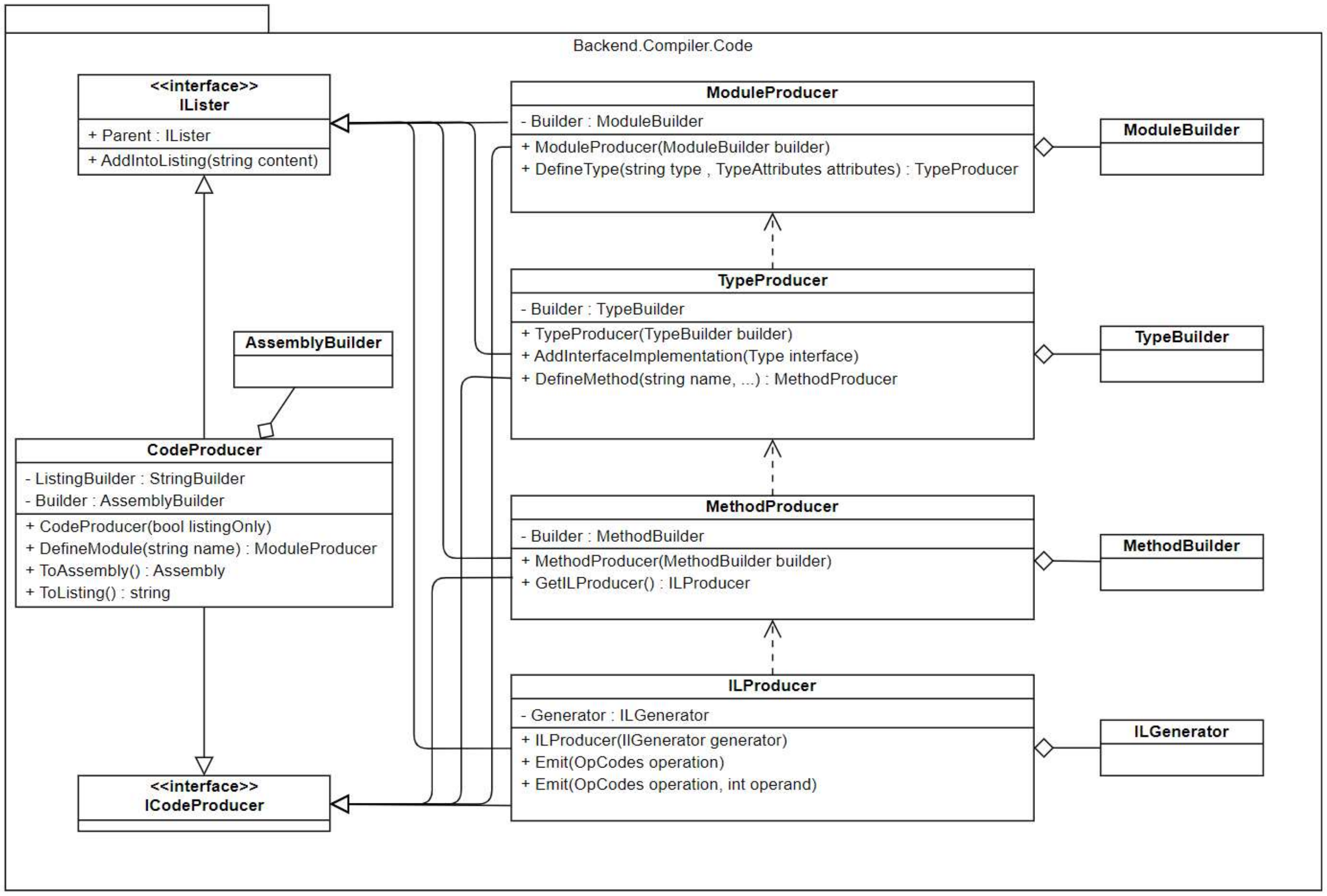
4.16. CallStandardGenerator

This class generates call statements for standard procedures.

5. Code Producer Design

The main intent for Code Producer design is providing ability to generate human being readable listing of CLR code generated.   The approach in design is to cover classes for .NET Emit namespace with corresponding decorators and implement listing capturing code in there. The `CodeProducer` calls is a decorator for `AssemblyBuilder` from .NET Emit namespace, the `ModuleProducer` the same for `ModuleBuilder` and so on.

All decorators must implement `ILister` interface to capture listing of the CLR code generated. All listing lines are collected in `CodeProducer` with `ListingBuilder` property instance. The other classes should propagate listing content to the `Parent`.


Backend.Compiler.Code
<<interface>>
ILister
+ Parent : ILister
+ AddIntoListing(string content)
ModuleProducer
- Builder : ModuleBuilder
+ ModuleProducer(ModuleBuilder builder)
+ DefineType(string type , TypeAttributes attributes) : TypeProducer
ModuleBuilder
TypeProducer
- Builder : TypeBuilder
+ TypeProducer(TypeBuilder builder)
+ AddInterfaceImplementation(Type interface)
+ DefineMethod(string name, ...) : MethodProducer
TypeBuilder
AssemblyBuilder
CodeProducer
- ListingBuilder : StringBuilder
- Builder : AssemblyBuilder
+ CodeProducer(bool listingOnly)
+ DefineModule(string name) : ModuleProducer
+ ToAssembly() : Assembly
+ ToListing() : string
MethodProducer
- Builder : MethodBuilder
+ MethodProducer(MethodBuilder builder)
+ GetILProducer() : ILProducer
MethodBuilder
ILProducer
- Generator : ILGenerator
+ ILProducer(IlGenerator generator)
+ Emit(OpCodes operation)
+ Emit(OpCodes operation, int operand)
ILGenerator
<<interface>>
ICodeProducer

Here is detailed breakdown of the class diagram.

5.1. ILister

This interface is implemented by classes that need to capture a listing of the generated CLR code.

Properties:

- **Parent -** allows content propagation to the parent lister.

Methods:

- **AddIntoListing -** A method that adds the provided content into the listing.

5.2. ICodeProducer

This interface is implemented by classes that produce code.

5.3. CodeProducer

This class implements both `ILister` and `ICodeProducer` interfaces. It serves as a decorator for the `AssemblyBuilder` class from the .NET Emit namespace.

Properties:

- **ListingBuilder -** A private property that collects all lines of the listing.
- **Builder -** A private property of type `AssemblyBuilder` . Used for calls delegation.

Methods:

- **CodeProducer -** A constructor method that initializes a new instance of the `CodeProducer` class.
- **DefineModule -** A method that returns an instance of `ModuleProducer` .
- **ToAssembly -** A method that returns an `Assembly` .
- **ToListing -** A method that returns a string representation of the listing.

5.4. ModuleProducer

This class implements both `ILister` and `ICodeProducer` interfaces. It serves as a decorator for the `ModuleBuilder` class from the .NET Emit namespace.

Properties:

- **Builder -** A private property of type `ModuleBuilder`. Used for calls delegation.

Methods:

- **ModuleProducer -** A constructor method that initializes a new instance of the `ModuleProducer` class.
- **DefineType -** A method that returns an instance of `TypeProducer`.

5.5. TypeProducer

This class implements both `ILister` and `ICodeProducer` interfaces. It serves as a decorator for the `TypeBuilder` class from the .NET Emit namespace.

Properties:

- **Builder -** A private property of type `TypeBuilder`. Used for calls delegation.

Methods:

- **TypeProducer -** A constructor method that initializes a new instance of the `TypeProducer` class.
- **AddInterfaceImplementation -** A method that adds an interface implementation to the type.
- **DefineMethod -** A method that returns an instance of `MethodProducer`.

5.6. MethodProducer

This class implements both `ILister` and `ICodeProducer` interfaces. It serves as a decorator for the `MethodBuilder` class from the .NET Emit namespace.

Properties:

- **Builder -** A private property of type `MethodBuilder`. Used for calls delegation.

Methods:

- **MethodProducer -** A constructor method that initializes a new instance of the `MethodProducer` class.
- **GetILProducer -** A method that returns an instance of `ILProducer`.

5.7. ILProducer

This class implements both `ILister` and `ICodeProducer` interfaces. It serves as a decorator for the `ILGenerator` class from the .NET Emit namespace.

Properties:

- **Generator -** A private property of type `ILGenerator` .

Methods:

- **ILProducer -** A constructor method that initializes a new instance of the `ILProducer` class.
- **Emit -** A overload method that emits the specified CLR operation.

6. Readable Representation

To facilitate debugging and ensure testability, the OML compiler Execution Flow Generator should be capable of producing human-readable code listings. The example below uses complex nested structures as a case study to demonstrate how the generated code could be presented. Please note that the output format is not set in stone and may be modified or expanded as necessary during the development process.

OML Code Sample

```
DECLARE SUB MyMethod ()

CALL MyMethod

SUB MyMethod
    FOR loopCounter = 1 TO 5
        PRINT loopCounter
    NEXT loopCounter
END SUB
```

IL Listing

```
.assembly MyAssembly {}
.module MyModule

.class public auto ansi beforefieldinit MyType extends [mscorlib]System.Object
{
  .method public hidebysig static void  MyMethod() cil managed
  {
    .maxstack  2
    .locals init ([0] int32 loopCounter)

    ldc.i4.1
    stloc.0

  IL_0002:
    ldloc.0
    call       void [mscorlib]System.Console::WriteLine(int32)
    ldloc.0
    ldc.i4.1
    add
    stloc.0
    ldloc.0
    ldc.i4.5
    ble.s      IL_0002
```

```
      ret
    }

    .method public hidebysig specialname rtspecialname instance void  .ctor() cil managed
    {
      .maxstack  8
      ldarg.0
      call       instance void [mscorlib]System.Object::.ctor()
      ret
    }
  }
```

7. Implementation notes

This design should be implemented in `OCLCMacroLanguage` project in `OCLCMacroLanguage.Backend.Compiler` namespace.

The initial implementation of `OCLCMacroLanguage.Backend.Compiler.Generators` requires only stub classes implementation. The actual code will be added during DSL language features implementation.

The initial implementation of `OCLCMacroLanguage.Backend.Compiler.Code` namespace requires implementation of all features presented in the design, however it is expected to be extended in the future development with more functionality.

No labels

4 Comments

Montgomery,Matt
I'm rusty on UML of this complexity.
What does the Generate interface method return?

Radzivilo(Contractor),Viktor
The void type is assumed. Explicitly mentioned in the method textual description.

Loesch,Tommy
Is there a good place to read more about the " .NET Emit namespace" before the Thursday meeting?

Radzivilo(Contractor),Viktor
Here is a MS document Emitting Dynamic Methods and Assemblies - .NET Framework | Microsoft Learn

Also, code generation with .NET Emit namespace is described in POC materials:

POC Native CLR code generation - Connexion Client - OCLC Confluence

Added hyperlink to #4 as well.