(1) Semantics of Inheritance and Dynamic Dispatch

Inheritance semantics:

- Classes can inherit from a parent class using the `parent` parameter in `ClassDef`.
- Subclasses inherit variables and methods from their superclasses unless they override them.
- Variable shadowing is allowed; a subclass can declare a variable with the same name as one in its superclass, effectively hiding the superclass's variable.

Dynamic Dispatch Semantics:

- Method calls are resolved at runtime based on the actual class of the instance
- When a method is invoked on an instance, the method lookup starts from the instance's class and moves up the inheritance hierarchy until the method is found

Here's an example where the Child class overrides the "greet" method from Parent:

```
// Define the Parent class
eval(ClassDef("Parent",
 body = Perform(List(
   MethodDef("greet", List(), Value("Hello from Parent"))
 ))
))

// Define the Child class that extends Parent
eval(ClassDef("Child",
 body = Perform(List(
   MethodDef("greet", List(), Value("Hello from Child"))
 )),
 parent = Some("Parent")
))

// Create instances of Parent and Child
eval(CreateInstance("Parent", "p"))
eval(CreateInstance("Child", "c"))

// Invoke greet on Parent instance 'p'
val parentGreet = eval(InvokeMethod("p", "greet", List()))
println(s"Parent instance 'p' says: $parentGreet") // Expected output: "Hello
from Parent"

// Invoke greet on Child instance 'c'
val childGreet = eval(InvokeMethod("c", "greet", List()))
println(s"Child instance 'c' says: $childGreet") // Expected output: "Hello
from Child"
```

(2) OOP Language Extension

- The language includes constructs for:
  - **Class Definitions (`ClassDef`):** To define new classes, optionally specifying a parent class for inheritance.
  - **Class Variables (`ClassVar`):** Declare variables within a class.
  - **Method Definitions (`MethodDef`):** Define methods with parameters and bodies.
  - **Instance Creation (`CreateInstance`):** Create new instances of classes.
  - **Method Invocation (`InvokeMethod`):** Invoke methods on class instances.
  - **Nested Classes:** Allow classes to be defined within other classes.
  - **Variable and Assignment Handling:** Manage variables within different scopes (method, instance, class)

Here's an example of where to use some of these constructs:

```
// Define the Base class

eval(ClassDef("Base",

 body = Perform(List(

   ClassVar("x", "int"),

   MethodDef("getX", List(), Variable("x")),

   MethodDef("setX", List(Parameter("value", "int")), Assign("x",
Variable("value")))

 ))

))

// Define the Derived class that extends Base

eval(ClassDef("Derived",

 body = Perform(List(

   ClassVar("x", "int"), // Shadows Base.x

   MethodDef("getX", List(), Variable("x")), // Overrides Base.getX

   MethodDef("setX", List(Parameter("value", "int")), Assign("x",
Variable("value")))

 )),

 parent = Some("Base")))
```

(3) Logic for Evaluating Method Invocations on Class Instances

In the `eval` function, `InvokeMethod` case to evaluate method invocations.
Process:

- **Method Lookup:** Use `lookupMethod` to find the method definition, starting from the instance's class and moving up the inheritance hierarchy. This helper functions is found in the HelperFunctions.scala file
- **Parameter Binding:** Bind the provided arguments to the method's parameters, creating a method environment.
- **Environment Stack:** Push the method environment and the instance environment onto the environment stack.
- **Method Evaluation:** Evaluate the method body within the combined environment (method and instance).
- **Environment Cleanup:** Pop the environments from the stack after evaluation.

Example of method invocation on an instance:

```
// Create an instance of Derived
eval(CreateInstance("Derived", "d"))

// Set x in Derived instance 'd'
eval(InvokeMethod("d", "setX", List(Value(10))))
```

(4) Implementation of Nested Classes with Scoping Rules for Obscuring and Shadowing

Nested Classes:

- Frizzy allows classes to be defined within the body of another class using `ClassDef`.
- Nested classes are registered in the global `classRegistry` to allow for instance creation and method invocation.

Scoping Rules:

- **Variable Shadowing:**
    - Variables declared in a nested class can have the same name as variables in the outer class.
    - When a variable is accessed within a method, the lookup follows this order:
        1. Method Environment (local variables and parameters)
        2. Instance Environment (instance variables)

3. Outer Class Environments (not currently implemented but can be extended)

- **Obscuring Variables:**
  - A variable in an inner scope can obscure a variable with the same name in an outer scope.
  - In the current implementation, inner classes do not have access to the variables of their outer classes unless explicitly passed or through additional mechanisms.

Example:

```
// Define the Outer class with a nested Inner class

eval(ClassDef("Outer",

 body = Perform(List(

   ClassVar("x", "int"),

   MethodDef("getX", List(), Variable("x")),

   // Define the nested Inner class

   ClassDef("Inner",

     body = Perform(List(

       ClassVar("x", "int"), // Shadows Outer.x

       MethodDef("getX", List(), Variable("x"))

     ))

   )

 ))

))
```