一、代码结构与设计

1. 变量的管理

```
@Derive[ToString]
public enum Binding {
    | LET_Uninitialized(Value)
    | VAR_Uninitialized(Value)
    | VAR_Initialized(Value)
    | LET_Initialized(Value)

    public static func letFrom(value: Value): Binding {
        Binding.LET_Initialized(value)
    }

    public static func varFrom(value: Value): Binding {
        Binding.VAR_Initialized(value)
    }

    public static func getValue(binding: Binding): Value {
        match (binding) {
            case Binding.LET_Uninitialized(_) => throw Exception("Trying to access uninitialized let binding"
            case Binding.VAR_Uninitialized(_) => throw Exception("Trying to access uninitialized var binding"
            case Binding.LET_Initialized(v) => v
            case Binding.VAR_Initialized(v) => v
        }
    }
}
```

使用对象Binding,其四个枚举值分别对应LET未初始化变量、VAR未初始化变量、LET初始化对象、VAR初始化对象，方便处理之后的赋值语句和变量声明语句。

```
// 无初始值：根据关键字与类型标注处理
case None =>
    if (isLet) {
        if (context.getLocal(name).isSome()) {
            throw CjcjRuntimeErrorWithLocation(ErrorCode.DUPLICATED_DEF,
                "redeclared variable `${name}`", decl)
        } else {
            match (decl.declType) {
                case Some(typeNode) =>
                    match (getAnnotatedTypeName(typeNode)) {
                        case "Int64" => context.declare(name, Binding.LET_Uninitialized(Value.VInteger(0)))
                        case "String" => context.declare(name, Binding.LET_Uninitialized(Value.VString("")))
                        case "Bool" => context.declare(name, Binding.LET_Uninitialized(Value.VBoolean(false)))
                        case "Unit" => context.declare(name, Binding.LET_Uninitialized(Value.VUnit))
                        case _ =>
                            throw CjcjRuntimeErrorWithLocation(ErrorCode.DEF_TYPE_MISMATCH,
                                "unsupported type annotation for uninitialized constant `${name}`", decl)
                    }
                case None =>
                    context.declare(name, Binding.LET_Uninitialized(Value.VUnit))
```

在变量声明语句中，即使没有赋值，UInitiated也会存储各种类型的默认值，这样之后赋值的时候便可以确认其是否定义了固定的类型

2. 作用域的管理

```
// 当前的静态作用域
private var context = Environment<String, Binding>()
// 全部的静态作用域
private var static_scope = ArrayList<Environment<String, Binding>>()
```

用context表示当前所处的作用域，用static_scope表示所定义的所有的作用域，用Environment的enclosing便可以从当前的context向上遍历到所有父作用域

3. 跳转语句的实现

```
// 当前所处循环的层数，处理 break / continue 的合法性
private var loopDepth: Int64 = 0
```

用全局变量loopDepth记录当前所处嵌套深度，在访问while语句的时候对其值进行修改

```
public open override func visit(expr: WhileExpr): Value {
    var cond = expr.condition.traverse(this)

    match (cond) {
        case VBoolean(b) =>
            loopDepth += 1
            var check = b
            while (check) {
                // 新建作用域
                var new_scope = Environment<String, Binding>(Some(context))
                static_scope.add(new_scope)
                context = new_scope
                var breaking = false
                try {
                    expr.block.traverse(this)
                } catch (_: BreakSignal) {
                    breaking = true
                } catch (_: ContinueSignal) {
                    // continue to next iteration
                } catch (e: Exception) {
                    if (let Some(enclosing) <- new_scope.enclosing) {
                        context = enclosing
                    }
                    loopDepth -= 1
                    throw e
                }
                // 恢复上下文
```

定义两个Exception用于实现break和continue的跳转，其中continue只需要打断并继续重新循环即可，break则需要直接跳出循环。

```
private class BreakSignal <: Exception {}

private class ContinueSignal <: Exception {}
```

4. 工具函数的实现

①判断TypeNode所处的类型

```
private func getAnnotatedTypeName(typeNode: TypeNode): String {
    // 原生类型：Bool、Unit、各数值类型(Int8/16/32/64/…)
    if (let Some(p) <- (typeNode as PrimitiveType)) {
        let tk = p.typeName.kind
        match (tk) {
            case TokenKind.INT8 => "Int8"
            case TokenKind.INT16 => "Int16"
            case TokenKind.INT32 => "Int32"
            case TokenKind.INT64 => "Int64"
            case TokenKind.BOOLEAN => "Bool"
            case TokenKind.UNIT => "Unit"
            case _ => "UnknownPrimitiveType"
        }
    // 引用类型：这里处理 String
    } else if (let Some(r) <- (typeNode as RefType)) {
        let tn = r.typeName.value
        match (tn) {
            case "String" => "String"
            case _ => "UnknownRefType"
        }
    } else {
        "UnknownType"
    }
}
```

② 判断TypeNode与Value是否类型相同

```
private func isSameTypeAnnotationAndValue(typeNode: TypeNode, value: Value): Bool {
    // 原生类型：Bool、Unit、各数值类型(Int8/16/32/64/…)
    if (let Some(p) <- (typeNode as PrimitiveType)) {
        let tk = p.typeName.kind
        match (value) {
            // 本运行时只有 VInteger(Int64)，把所有“数值型”都视为可接受
            case VInteger(_) =>
                tk == TokenKind.INT64
            case VBoolean(_) =>
                tk == TokenKind.BOOLEAN
            case VUnit =>
                tk == TokenKind.UNIT
            case VString(_) =>
                false
        }
    // 引用类型：这里处理 String
    } else if (let Some(r) <- (typeNode as RefType)) {
        let tn = r.typeName.value
        match (value) {
            case VString(_) => tn == "String"
            case _ => false
        }
    } else {
        false
    }
}
```

③判断if语句是否有else分支

```
private func isDefaultElseBranch(branch: Expr): Bool {
    if (let Some(block) <- (branch as Block)) {
        let blockNodes = block.nodes
        if (blockNodes.size == 1) {
            let firstNode = blockNodes[0]
            if (let Some(literal) <- (firstNode as LitConstExpr)) {
                return literal.literal.kind == TokenKind.UNIT_LITERAL
            }
        }
    }
    false
}
```

④ 递归向上覆盖最早的定义name的变量，将值修改为Binding，用于处理赋值语句

```
private func setValueInContext(name: String, binding: Binding): Bool {
    var env = context
    while (true) {
        if (env.elements.contains(name)) {
            env.elements.remove(name)
            env.elements.add(name, binding)
            return true
        }
        env = match (env.enclosing) {
            case None => return false
            case Some(e) => e
        }
    }
    false
}
```

⑤一些简单的逻辑判断函数

```
private func isSameValueType(lhs: Value, rhs: Value): Bool {
    match ((lhs, rhs)) {
        case (VInteger(_), VInteger(_)) => true
        case (VString(_), VString(_)) => true
        case (VBoolean(_), VBoolean(_)) => true
        case (VUnit, VUnit) => true
        case _ => false
    }
}

private func isUnit(value: Value): Bool {
    match (value) {
        case VUnit => true
        case _ => false
    }
}
```

二、遇到的困难与解决方案

1. 如何确定一个变量是否已经赋值、是否已经定义类型、是否只能赋值一次

解决方案：定义binding类型来区分不同类型变量

2. 如何处理continue和break的跳转逻辑

解决方案：定义新的Exception来实现中断退出，遇到continue和break都抛出Exception

3. 如何确定一个VarDecl是否有确定固定的类型

解决方案：实现getAnnotatedTypeName来获取TypeNode的对应类型

三、 已知BUG

暂时没有