Lab2 实验报告

**函数、高阶作用域与闭包执行器实现**

# 1. 目标与需求

在 Lab1 的基础上，本次 Lab2 实现了：

| 功能点 | 描述 | 完成情况 |
| --- | --- | --- |
| 函数定义 `func` | 能够声明函数及返回类型 | 已实现 |
| 函数调用 `f(x,y)` | 支持实参求值、形参绑定与作用域隔离 | 已实现 |
| Return 语义 | 支持 `return expr` 并正确退出函数 | 已实现 |
| 变量作用域扩展 | 支持 **全局/局部作用域**，递归进入与退出 | 已实现 |
| 闭包与环境捕获 | 函数可携带外层作用域变量运行 | 已实现 |
| 函数可先调用后声明 | 实现 **函数 Hoisting**（变量仍不 hoist） | 已实现 |
| While、If、Block 等表达式支持 | 继承 Lab1，已与函数执行整合 | 已实现 |

# 2. 关键设计与实现说明

## 2.1 Value 类型扩展

Lab1 仅支持原始类型，为支持函数运行，新拓展：

```
public enum Value {
    | VInteger(Int64)
    | VString(String)
    | VBoolean(Bool)
    | VUnit
    | VFunction(UserFunction)               // 用户自定义函数
    | VBuiltinFunction(BuiltinFunction)     // 内建函数 println
}
```

- 使 `CallExpr` 运行成为可能
- 允许闭包持值并在运行时动态调取

## 2.2 VarInfo 扩展（保存函数与全局状态）

```
public class VarInfo {
    public let isMutable: Bool
    public let declaredType: ?PrimitiveType
    public var initialized: Bool
    public var value: Value
    public let isGlobal: Bool               // ★ 用于闭包可见性
}
```

改进点：

| 字段 | 作用 |
| --- | --- |
| `initialized` | 防止使用未初始化变量 |
| `isGlobal` | 用于判断是否在闭包中可见 |
| `value` | 允许储存 `VFunction` 类型 |

## 2.3 函数提升（Hoisting）实现核心

目标：**允许函数调用先于声明**

实现思路：在执行前 **扫描两遍 Program**

```
        for (d in prog.decls.iterator()) {
            match (d) {
                // 顶层函数定义：调用你已经写好的 visit(FuncDecl)
                case f: FuncDecl =>
                    f.traverse(this)
                case _ =>
                    ()
            }
        }

        for (d in prog.decls.iterator()) {
            match (d) {
                // 记录 main，但先不执行
                case m: MainDecl =>
                    mainDeclOpt = Some(m)

                case v: VarDecl =>
                    v.traverse(this)

                case _ =>
                    ()
            }
        }
```

## 2.4 函数调用执行流程 CallExpr

新增

```
case Value.VFunction(f) =>
                return this.callUserFunction(f, call)
```

```
// 供 CallExpr 使用的函数调用实现
    private func callUserFunction(fn: UserFunction, call: CallExpr): Value {
        // 1. 参数数量检查
        let expected = fn.params.size
        let actual = call.arguments.size
        if (expected != actual) {
            throw CjcjRuntimeErrorWithLocation(
                ErrorCode.CALL_ARG_COUNT_MISMATCH,
```

```
                "Function `${fn.name}` expects ${expected} arguments, but got
${actual}",
                call
            )
        }

        let argValues = ArrayList<Value>()
        var idx: Int64 = 0
        for (arg in call.arguments.iterator()) {
            // 如果 Argument 里是 'expr' 字段，就用 arg.expr.traverse(this)
            // 如果你已经实现了 visit(Argument)，也可以直接 arg.traverse(this)
            let v = arg.traverse(this)
            argValues.add(v)

            // 形参类型检查（如果 RuntimeParam 里带了 type）
            let rp = fn.params[idx]
            if (rp.decltype.isSome()){
                let tyNode = rp.decltype.getOrThrow()
                match (tyNode) {
                    case t: PrimitiveType =>
                        let declared = t.typeName.value
                        let actualName = this.valType(v)
                        if (declared != actualName) {
                            throw CjcjRuntimeErrorWithLocation(
                                ErrorCode.CALL_ARG_TYPE_MISMATCH,
                                "parameter ${idx} of function `${fn.name}`
expects ${declared}, got ${actualName}",
                                arg              // 具体的实参位置
                            )
                        }
                    case _ =>
                        ()
                }
            }

            idx = idx + 1
        }

        let oldScopes = this.scopes
        let oldFunction = this.currentFunction
        this.scopes = ArrayList<HashMap<String, VarInfo>>()
        for (scope in fn.capturedScopes.iterator()) {
            this.scopes.add(scope)
        }

        this.currentFunction = Some(fn)

        this.pushScope()
        let paramScope = this.currentScope()

        // 4. 把形参名 -> 实参值 绑定到 paramScope 中（不可变 var）
        idx = 0
        while (idx < fn.params.size) {
            let rp = fn.params[idx]
            let argVal = argValues[idx]

            let paramName = rp.name
            paramScope[paramName] = VarInfo(
```

```
                false,              // 参数不可变
                None,       // 简化：不存 PrimitiveType，靠 valType 检查
                true,
                argVal,
                false
            )

            idx = idx + 1
        }

        //执行函数体，捕获 ReturnSignal
        var retVal: Value = Value.VUnit
        try {
            retVal = fn.body.traverse(this)
        } catch (e: ReturnSignal) {
            retVal = e.value
        }

        //调用结束，弹出参数作用域，并恢复之前的 scopes和函数上下文
        this.popScope()
        this.scopes = oldScopes
        this.currentFunction = oldFunction

        if (fn.returnType.isSome()) {
            let retTyNode = fn.returnType.getOrThrow()
            match (retTyNode) {
                case t: PrimitiveType =>
                    let declaredRet = t.typeName.value
                    let actualRet = this.valType(retVal)
                    if (declaredRet != actualRet) {
                        throw CjcjRuntimeErrorWithLocation(
                            ErrorCode.FUNC_RETURN_TYPE_MISMATCH,
                            "function `${fn.name}` declared return type
${declaredRet}, but returned ${actualRet}",
                            call
                        )
                    }
                case _ =>
                    ()
            }
        }
        return retVal
    }
```

## 2.5 闭包实现 (Closure Scoping)

闭包（Closure）是本次 Lab2 的核心能力，它允许函数在创建时记录当前作用域环境，即使在之后执行，也仍然能够访问当时的变量。

首先我们对`UserFunction`结构扩展为包含环境

```
public class UserFunction {
    name:String
    params:ArrayList<RuntimeParam>
    returnType:?TypeNode
    body:Block
    capturedScopes:ArrayList<HashMap<String,VarInfo>>   // 关键
    defOrderInParent:Int64
}
```

| 字段 | 作用 |
| --- | --- |
| capturedScopes | 保存函数创建时整个作用域链，闭包运行依据此环境执行 |
| body | 闭包执行时实际运行体 |
| params | 运行时创建参数作用域并置于最顶层 |

函数执行时恢复 capturedScopes

```
old = scopes
scopes = fn.capturedScopes  // 恢复函数定义时的环境
pushScope()                 // 新建参数作用域（局部变量区）

执行 body / 若遇 return 抛 ReturnSignal

popScope()
scopes = old                // 结束后恢复原作用域
```

# 3. 实现中的问题与解决方案

## 问题1：函数先调用再声明时报 UNDEFINED_VAR

出现：

```
let a = add(1,2)
func add(x,y){ ... }
```

原因：Lab1 环境不支持函数提前使用。

解决方案：
**Program 先扫描一遍顶层函数声明进行 Hoisting 占位注册**，再遍历一遍生成真实函数对象。生成针对lab2的executor

成果：函数可先调用后声明，但变量依然必须先定义后使用，保持语言一致性。

## 问题2：闭包调用中作用域链丢失

初期执行：

```
var g = 1
func f(){ println(g) }
f()  // → undefined variable 'g'
```

原因：调用函数时新建的作用域覆盖了原作用域，找不到 g。

解决：

- 在 FuncDecl 构造 UserFunction 时使用 `copyScopesForClosure()`
- 调用时恢复 capturedScopes

闭包访问外部变量成功。

## 问题3：Return 无法中断 Block 执行

现象：

```
func f(){ return 3 ; println("never") }
```

但还会继续执行 println。

原因：没有 ReturnSignal 异常机制。

解决：

```
throw ReturnSignal(value)
```

在 callUserFunction 捕获后返回，完全与真实语言一致。

## 问题4：实参求值与形参绑定顺序出错

初版 CallExpr 实现如下：

```
for arg in call.arguments:
    bindParam(arg)   // ❌ 未先求值
```

导致闭包使用错误环境。

修复：

```
for arg in arguments:
    value = arg.traverse(this)  // ✓先求值
    paramScope[name] = VarInfo(value)
```

**Call 已严格遵守 expr evaluate → bind param → execute function 流程**

# 4. Debug 与踩坑记录

| Bug内容 | 原因 | 解决方式 |
| --- | --- | --- |
| `+= ++` 等语法报 expected '}' | 仓颉 **不支持复合赋值** | 统一替换为 `x = x + 1` |
| 死循环 while 中 x 未增长 | 缺少 `x = x + 1` | 加入后循环正确退出 |
| 闭包修改变量未生效 | VarInfo 未共享引用 | capturedScopes 中保存 VarInfo 引用而非复制 |
| 多次声明同名函数未报错 | 缺少重复检查 | 在 Hoisting Pass1 检测重名立即抛错 |

Debug 完整解决，闭包可用、递归正常、所有测试均通过。