PA2 实验报告

计 73 周炫柏 2017011460

实验步骤：

一.合并框架

1.使用了 meld 进行代码的合并工作，完成简单的合并（包括关键字的注册、新类的声明、jacc 文件中表达式的添加等）任务后，可以通过 PA1-A 所有的测试点。

2.试运行 PA2 的测试点时，发现报了许多 ERROR，询问同学后发现需要解决 nullpointererror 的问题。对有可能是 null 的变量需要进行判断后再进行访问。

二.本次试验主要思路梳理

本阶段的任务是对 Decaf 程序进行语义分析，采取多次遍历 AST 的方法。采用访问者模式，主要分析逻辑在 Namer.java 和 Typer.java 中完成，Tree.java 中定义的类型需要进行语义分析的时候调用 accept 函数。

三.主要实验部分

1.添加 ABSTRACT 关键字：

a.前期工作

1°已完成添加关键字、更改表达式，下面只需要进行 PA2 附加部分的编写。

2°在 ClassDef 类中添加指示类是否为抽象类的参数 isAbstract，在建立 ClassSymbol 类的时候，访问该值并由此判断这个 Symbol 对应的类是否为抽象类。

3°修改给出的 ClassSymbol 类，在输出函数 str 中新增一条判断语句，如果是抽象类则在输出语句的最前端添加关键字 ABSTRACT。

b.错误检查：

1°Main 类不能为抽象类

在 Namer.java 中修改 VisitTopLevel 函数，添加①类是否名为 Main，②类是否为抽象类的判断逻辑。如果某个类被定义为 Main 且为抽象类，需要报错 NoMainClassError。

```
//检查是否存在定义Main类为Abstract的错误
for (var clazz : program.classes) {
    if (clazz.isAbstract && clazz.name.equals("Main")) {
        issue(new NoMainClassError());
        break;
    }
}
```

2°含有抽象成员的类不能被声明为非抽象类

按照实验指导书的建议，在 ClassSymbol 中维护一个 AbstractMethodList（类型为 List<String>）用于记录当前类中的所有抽象成员。对这个 List 进行的操作有：①增加：如果当前类新定义了一个抽象成员，或者是继承了父类的抽象成员，则将这些抽象成员加入 List 当中；②删除：如果当前类的某个方法在父类中事先被声明过了（即子类在试图重载父类的方法），且在该类中这个方法为非抽象的，在父类中对应的方法

为抽象的，则将该成员从 List 中删去；③遍历：最终在 VisitClassDef 中遍历这个 AbstractMethodList，如果该类被声明为非抽象的但是这个 List 不为空，需要报错。具体来说：

a.删除操作：在 VisitMethodDef 类中进行修改。在该类判断是否成功重载的函数体内加入，如果成功重载了父类的抽象方法，则将方法名从 List 中删去的操作。

```
//如果该方法不是抽象的，而父类的方法是抽象的，则说明子类成功重载了父类的抽象方法，应在子类的abstractmethodlist中删去该方法
if (!method.isAbstract() && suspect.isAbstract) {
    ctx.currentClass().abstractMethodList.remove(method.name);
}
```

b.添加操作：在 VisitMethodDef 类中进行修改。如果这个方法未在父类中出现过，且该方法为抽象的，则将该方法加入到 List 中。

```
//若这个抽象方法之前没有定义过（父类没有对应方法），则表示是子类新建的抽象方法，需加入子类的abstractmethodlist
if(method.isAbstract()) {
    method.symbol.isAbstract = true;
    ctx.currentClass().abstractMethodList.add(method.name);
}
```

c.遍历操作：在 VisitClassDef 类中进行修改。如果某个类被声明为非抽象的但它的 AbstractMethodList 不为空，需要报错。

```
// class 不是抽象类且 abstarctmethodlist非空
if (!clazz.isAbstract && !clazz.symbol.abstractMethodList.isEmpty()) {
    issue(new AbstractClassNotSatisfied(clazz.pos, clazz.name));
}
```

3°已经被定义的非抽象类不能被再次声明为抽象类:

在上述提到的判断是否成功重载的函数体内新增判断，如果子类继承自父类非抽象的方法被试图重新定义为抽象的，需要报错。

```
//如果该方法是抽象的，而父类的方法是非抽象的，代表子类错误地重载了父类的方法，需报错
if (method.isAbstract() && !suspect.isAbstract) {
    issue(new DeclConflictError(method.pos, method.name, earlier.get().pos));
}
```

4°不能 New 一个抽象类。

在 VisitNew 类中进行判断。如果被 New 的类为抽象类，报错。

```
@Override
public void visitNewClass(Tree.NewClass expr, ScopeStack ctx) {
    var clazz = ctx.lookupClass(expr.clazz.name);
    if (clazz.isPresent()) {
        //如果新建的这个类不是抽象的，则正常声明
        if(!clazz.get().isAbstract) {
            expr.symbol = clazz.get();
            expr.type = expr.symbol.type;
        } else {//否则需要报错
            issue(new NewAbstractMainError(expr.pos, expr.clazz.name));
            expr.type = BuiltInType.ERROR;
        }
    } else {
        issue(new ClassNotFoundError(expr.pos, expr.clazz.name));
        expr.type = BuiltInType.ERROR;
    }
}
```

至此完成了 ABSTRACT 相关部分代码的编写。

2.局部类型推断：
a.前期工作：
1°添加关键字 VAR，新增表达式，添加相关文件。

b.错误检查：
1°如果推导出的类型为 void，需要报错。
在 VisitLocalVarDef 类中添加判断逻辑，如果推断出表达式右侧的类型为 void，报错。

```
var lt = stmt.symbol.type;
var rt = initVal.type;
if (lt == null) {//左侧部分没有定义类型
    if (rt.isVoidType()) {//右侧类型被推断为void
        issue(new BadVarTypeError(stmt.id.pos, stmt.id.name));
        stmt.symbol.type = BuiltInType.ERROR;
        return;
    } else {//右侧类型是正常类型
        stmt.symbol.type = rt;
    }
} else {//左侧部分已声明类型
    if (lt.noError() && (!rt.subtypeOf(lt))) {//如果右边类型不能和左侧兼容，报错
        issue(new IncompatBinOpError(stmt.assignPos, lt.toString(), "=", rt.toString()));
    }
}
```

2°部分 VAR 相关代码牵涉到 LAMBDA 表达式，在此先不做解释。

至此，完成目前能做到的 VAR 关键词相关部分代码。

3.First-class Functions
A.函数类型
a.前期工作：
1°TLAMBDA 声明及类的定义，包括 visitTLambda 函数的定义，表达式的添加。

b.具体实现：
1°在 TypeLit.java 中新增 visitTLambda 函数，实现对 TLambda 表达式返回值和参数类型的解析和判误操作。

```java
@Override
default void visitTLambda(Tree.TLambda typeLambda, ScopeStack ctx) {
    // 解析返回值
    typeLambda.returnType.accept(this, ctx);
    if (typeLambda.returnType.type.eq(BuiltInType.ERROR)) {
        typeLambda.type = BuiltInType.ERROR;
    }
    // 解析参数
    var hasError = false;
    var typeList = new ArrayList<Type>();
    for (var param : typeLambda.typeList) {
        param.accept(this, ctx);
        if(param.type.eq(BuiltInType.ERROR)) {
            typeLambda.type = BuiltInType.ERROR;
            hasError = true;
        } else if (param.type.eq(BuiltInType.VOID)){//如果参数返回类型是void，则需要报错
            typeLambda.type = BuiltInType.ERROR;
            hasError = true;
            issue(new VoidArgError(param.pos));
        } else {
            typeList.add(param.type);
        }
    }

    if (!hasError) {
        typeLambda.type = new FunType(typeLambda.returnType.type, typeList);
    }
}
```

2°参数类型判断已在 VAR 中统一完成。

至此，该部分内容代码编写完成。

B.LAMBDA 表达式作用域
a.前期工作：
1°LAMBDA 在 KIND 表项中注册，构造 LAMBDA 类和对应的 VisitLambda 函数。
2°按照实验指导书的建议，新增两个 java 文件，LambdaScope.java 和 LambdaSymbol.java。定义一个 Lambda 表达式时，与定义函数类似，打开了一层新的参数作用域 FormalScope，存放各参数对应的变量符号，里面再是一层局部作用域 LocalScope（每个 LambdaScope 必有一个子 LocalScope）。LambdaSymbol 文件仿照 MethodSymbol，LambdaSymbol 仿照 LocalScope。

b.Lambda 表达式作用域类符号作用域检查
在 visitAssign 类中进行修改值的错误判断。已知判断错误的原则是：

1. 不能对捕获的外层的**非类**作用域中的符号**直接**赋值，但如果传入的是一个对象或数组的引用，可以通过该引用修改类的成员或数组元素。
2. 如果要将 Lambda 表达式赋值给一个**正在定义的**符号，则 Lambda 内部作用域中的变量既不能与该符号重名，也不能访问到该符号。

据此编写判断逻辑：

```java
@Override
public void visitAssign(Tree.Assign stmt, ScopeStack ctx) {
    stmt.lhs.accept(this, ctx);
    stmt.rhs.accept(this, ctx);
    var lt = stmt.lhs.type;
    var rt = stmt.rhs.type;
    // 不能对成员方法赋值
    if(stmt.lhs instanceof Tree.VarSel && ((Tree.VarSel)stmt.lhs).isMemberFuncName) {
        issue(new AssignToMemberMethodError(stmt.pos, ((Tree.VarSel)stmt.lhs).name));
    }

    // 错误类型判断，类型不兼容
    if (lt.noError() && (!rt.subtypeOf(lt))) {
        issue(new IncompatBinOpError(stmt.pos, lt.toString(), "=", rt.toString()));
    }
    if(lt.noError()){
        var currFuncScope = ctx.nearestFormalOrLambdaScope();
        if(currFuncScope.isLambdaScope()&& stmt.lhs instanceof Tree.VarSel){
            //直接赋值的情况（没有引用）
            if(((Tree.VarSel) stmt.lhs).receiver.isEmpty()){
                ListIterator<Scope> iter = ctx.scopeStack.listIterator(ctx.scopeStack.size());
                while(iter.hasPrevious()){
                    var scope =iter.previous();
                    if(scope == currFuncScope){
                        break;
                    }
                }
                while (iter.hasPrevious()){
                    var scope = iter.previous();
                    if(!scope.isClassScope() && ((Tree.VarSel) stmt.lhs).symbol.domain()==scope){
                        issue(new AssignToCapturedVarError(stmt.pos));
                        break;
                    }
```

在此处实现了所有不能修改值情况的检验。

c.解析 Lambda 表达式

实现逻辑是 Namer 中只完成参数类型检查，在 Typer 中进行返回类型推断。

1°Namer.java – VisitLambda 类 + TypeLambda 类。实现 Lambda 表达式的遍历，梳理出所有的参数类型并进行类型判断（使用到上面已经完成的部分逻辑）。

```java
@Override
public void visitLambda(Tree.Lambda lambda, ScopeStack ctx) {
    // lambda 表达式的返回类型此时不能确定故此时namer不建立symbol
    // 把第一次扫描的部分有用信息存在tree.Lambda的参数列表里
    var lambdaScope = new LambdaScope(ctx.currentScope());
    var localScope = new LocalScope(lambdaScope);
    // 检查参数列表
    typeLambda(lambda, ctx, lambdaScope);

    // 对lambda表达式后面的expr或block部分进行解析
    if (lambda.expr != null) {
        ctx.open(lambdaScope);
        ctx.open(localScope);
        lambda.expr.accept(this, ctx);
        ctx.close();
        ctx.close();
        // 用于记录，方便从Namer传递到Typer
        lambda.localScope = localScope;
    } else {
        ctx.open(lambdaScope);
        if (lambda.block != null){
            lambda.block.accept(this, ctx);
        }
        ctx.close();
    }
    // 用于记录，方便从Namer传递到Typer
    lambda.lambdaScope = lambdaScope;
}
```

```java
private void typeLambda(Tree.Lambda lambda, ScopeStack ctx, LambdaScope lambdaScope) {
    // 这里只做了参数类型检查，返回类型检查在typer中进行
    ctx.open(lambdaScope);
    var argTypes = new ArrayList<Type>();
    for (var param : lambda.varList) {
        param.accept(this, ctx);
        argTypes.add(param.typeLit.type);
    }
    // 用于记录，方便从Namer传递到Typer
    lambda.argTypes = argTypes;
    ctx.close();
}
```

2°返回类型推导：

根据实验指导书上的解释可以知道 Lambda 表达式返回类型的定义：

Lambda 表达式返回类型

你需要正确推导出 Lambda 表达式的返回类型。

记 $<:$ 是类型上的二元关系，它满足：

- 自反性：$t <: t$
- 传递性：If $t_1 <: t_2$ and $t_2 <: t_3$, then $t_1 <: t_3$
- 类继承：If c_1 extends c_2 , then $ClassType(c_1) <: ClassType(c_2)$
- 函数：If $t <: s$ and $s_i <: t_i$ for every i , then
  $FunType([t_1, t_2, \ldots, t_n], t) <: FunType([s_1, s_2, \ldots, s_n], s)$

对于 `fun (t1 x1, t2 x2, ...) => y` ，若推导出 `y` 的类型为 t ，那么整个 Lambda 表达式的类型为 $FunType([t_1, t_2, \ldots], t)$ 。

若 `y` 是一个 `BlockStmt` ，如果 `BlockStmt` 的所有执行路径都没有 `return` 语句，则 `y` 的类型是 `void` ；否则 `y` 的类型是所有 `return` 语句返回值类型的最小“上界”(无返回值的 `return` 语句可认为返回 `void` 类型)。定义：称 t 是类型 $t_1, t_2, \ldots, t_n$ 的上界，若 $t_1 <: t, t_2 <: t, \ldots, t_n <: t$ 。若该上界不存在，或返回类型不是 `void` 但存在一条执行路径无返回值，那么需要报错。

在此我们做一个总结：Lambda 表达式有后接 Expr 和 Block 两种不同内容的两种形式，两种形式的返回类型推断做法并不相同。对于前者，可以直接将 Expr 的类型 +Lambda 参数类型作为 Lambda 的返回类型，而后者则需要对 Block 再做返回类型推导。

对于 Block 的类型推导需要进行最小上界和最大下界的求解。

实现时，对每个 Lambda 表达式你需要先想办法得到它内部的所有 `return` 语句，然后可用递归的方式求它们类型的上界，下面给出一个参考算法：

求类型 $[t_1, t_2, t_3, \ldots, t_n]$ 的类型上界：

1. 选择其中一个非 `null` 的类型 t_k；
2. 如果 t_k 是基本类型(`int`，`bool`，`string`，`void`)或数组，检查其他类型是否与 t_k 完全等价，如果是返回 t_k，不是返回"类型不兼容"；
3. 如果 t_k 是 `ClassType`：
    1. 令 $p = t_k$，检查是否对所有 t_i 满足 $t_i <: p$，如果是返回 p，不是继续下面的操作；
    2. 令 $p = p$ 的父类；
    3. 如果 t_k 和其祖先都不是上界，返回"类型不兼容"；
4. 如果 t_k 是 `FunType`，先检查其他类型是否也都是 `FunType`，且形式与 t_k 相同，如果不是直接返回"类型不兼容"，否则：
    1. 设 $t_i = FunType([s_{i1}, s_{i2}, \ldots, s_{im}], r_i)$；
    2. 求 $[r_1, r_2, \ldots, r_n]$ 的类型**上界**，设其为 R；
    3. 求 $[s_{1i}, s_{2i}, \ldots, s_{ni}]$ 的类型**下界**，设其为 T_i；
    4. 返回 $FunType([T_1, T_2, \ldots, T_m], R)$。

根据参考算法得到对应的解决代码：

Ⅰ.求解最小上界 upbound

```
// 求 List<Type> T 的最小上界
    public Type upBound(List<Type> T) {
        Type type = null;
        // 选取非空 t_k
        for (var t : T) {
            if (!t.eq(BuiltInType.NULL)) {
                type = t;
                break;
            }
        }

        if (type == null || type.eq(BuiltInType.NULL)) {
            // 所有 return 类型均为 BuiltInType.NULL
            return BuiltInType.NULL;
        } else if (type.isVoidType() || type.isBaseType() || type.isArr
ayType()) {
            // INT BOOL STRING VOID 四个基本类型或数组
            for (var t : T) {
                // 如果存在 return 类型不相同，报类型不兼容错误
                if (!type.eq(t)) {
                    return BuiltInType.ERROR;
```

```
                }
            }
            // 所有类型相同，返回该类型
            return type;
        } else if (type.isClassType()) {
            // ClassType
            // 判断是否所有返回类型都 <: t_k
            while (true) {
                if (judgeMiniAncestor(T, type)){
                    // type 即为最小祖先
                    return type;
                } else {
                    if (((ClassType)type).superType.isPresent()){
                        // type 不是最小祖先但还有父类
                        type = ((ClassType)type).superType.get();
                    } else {
                        // type 不是最小祖先且没有父类
                        return BuiltInType.ERROR;
                    }
                }
            }
        } else {
            // FunType
            var funType =(FunType)type;
            var retList = new ArrayList<Type>();
            // 第 i 个成员为 所有 返回值的 argList 的第 i 个值 组成的 list
            var argLists = new ArrayList<ArrayList<Type>>();
            var argNum  = funType.arity();

            // 初始化 argLists
            for (int i = 0;i < argNum;i ++){
                argLists.add(new ArrayList<Type>());
            }

            for (var t : T) {
                // 检查所有参数类型是否相同，argNum 是否相同
                if (!t.isFuncType() || ((FunType)t).arity() != argNum)
{
                    return BuiltInType.ERROR;
                }

                // 整合返回值列表和参数列表
                var tt = (FunType)t;
                retList.add(tt.returnType);
```

```
                for (int i = 0;i < argNum; i ++){
                    argLists.get(i).add(tt.argTypes.get(i));
                }
            }
            // 求返回值上界
            var returnType = upBound(retList);
            if (returnType.eq(BuiltInType.ERROR)) {
                return BuiltInType.ERROR;
            }
            var retArgList = new ArrayList<Type>();
            for (int i = 0;i < argNum; i ++) {
                // 求参数列表中每个位置的参数的下界
                var arg = downBound(argLists.get(i));
                if (arg.eq(BuiltInType.ERROR)) {
                    return  BuiltInType.ERROR;
                }
                retArgList.add(arg);
            }
            return new FunType(returnType, retArgList);
        }
        // 所有返回类型为 BuiltInType.NULL，则返回类型为 NULL
    }

    public boolean judgeMiniAncestor(List<Type> T, Type type) {
        for (var t : T){
            if (!t.subtypeOf(type)) {
                return false;
            }
        }
        return true;
    }
```

II .求解最小下界 downBound

```
// 求 List<Type> T 的最大下界
    public Type downBound(List<Type> T) {
        Type type = null;
        // 选取非空 t_k
        for (var t : T) {
            if (!t.eq(BuiltInType.NULL)) {
                type = t;
                break;
            }
        }
```

```
        if (type == null || type.eq(BuiltInType.NULL)) {
            // 所有 return 类型均为 BuiltInType.NULL
            return BuiltInType.NULL;
        } else if (type.isVoidType() || type.isBaseType() || type.isArr
ayType()) {
            // INT BOOL STRING VOID 四个基本类型或数组
            for (var t : T) {
                // 如果存在 return 类型不相同，报类型不兼容错误
                if (!type.eq(t)) {
                    return BuiltInType.ERROR;
                }
            }
            // 所有类型相同，返回该类型
            return type;
        } else if (type.isClassType()) {
            // ClassType
            // 判断是否所有返回类型都 <: t_k
            for (var t:T) {
                if (t.subtypeOf(type)){
                    type = t;
                } else if (type.subtypeOf(t)) {
                    // do nothing
                } else {
                    // t 和 type 没有共同下界
                    return BuiltInType.ERROR;
                }
            }
            return type;
        } else {
            // FunType
            var funType =(FunType)type;
            var retList = new ArrayList<Type>();
            // 第 i 个成员为 所有 返回值的 argList 的第 i 个值 组成的 list
            var argLists = new ArrayList<ArrayList<Type>>();
            var argNum  = funType.arity();

            // 初始化 argLists
            for (int i = 0;i < argNum;i ++){
                argLists.add(new ArrayList<Type>());
            }

            for (var t : T) {
                // 检查所有参数类型是否相同，argNum 是否相同
```

```
                if (!t.isFuncType() || ((FunType)t).arity() != argNum)
{
                    return BuiltInType.ERROR;
                }

                // 整合返回值列表和参数列表
                var tt = (FunType)t;
                retList.add(tt.returnType);
                for (int i = 0;i < argNum; i ++){
                    argLists.get(i).add(tt.argTypes.get(i));
                }
            }
            // 求返回值下界
            var returnType = downBound(retList);
            if (returnType.eq(BuiltInType.ERROR)) {
                return BuiltInType.ERROR;
            }
            var retArgList = new ArrayList<Type>();
            for (int i = 0;i < argNum; i ++) {
                // 求参数列表中每个位置的参数的上界
                var arg = upBound(argLists.get(i));
                if (arg.eq(BuiltInType.ERROR)) {
                    return  BuiltInType.ERROR;
                }
                retArgList.add(arg);
            }
            return new FunType(returnType, retArgList);
        }
    }
```

III.完整的 visitLambda 代码

```java
@Override
public void visitLambda(Tree.Lambda lambda, ScopeStack ctx) {
    // 对expr或block进行遍历
    // 根据typecheck后的expr或block确定函数返回的类型
    if (lambda.lambdaScope == null) {
        System.out.println("FUCK");
    }
    ctx.open(lambda.lambdaScope);
    if (lambda.expr != null) {
        assert lambda.localScope != null;
        ctx.open(lambda.localScope);
        lambda.expr.accept(this, ctx);
        ctx.close();
        lambda.type = new FunType(lambda.expr.type, lambda.argTypes);
    } else {
        // 创建returnTypes List, 用于记录block accept的过程中所有返回的returnType
        // 然后在 getLambdaBlockReturnType 中根据此 list 进行类型推导
        lambda.lambdaScope.returnTypes = new ArrayList<>();
        lambda.block.accept(this, ctx);
        // 如果没有返回语句，return BuildInType.Void, 否则，进行最小上界类型推导
        Type blockReturnType = (lambda.lambdaScope.returnTypes.isEmpty()) ?
                BuiltInType.VOID : upBound(lambda.lambdaScope.returnTypes);
        if (!blockReturnType.isVoidType() && !lambda.block.returns) {
            issue(new MissingReturnError(lambda.block.pos));
        }
        if(blockReturnType.eq(BuiltInType.ERROR)) { //返回类型存在问题
            issue(new IncompatRetTypeError(lambda.block.pos));
        }
        lambda.type = new FunType(blockReturnType, lambda.argTypes);
        System.out.println("my Type: " + (FunType)lambda.type);
    }
    ctx.close();

    // 确定好函数返回类型，建立lambda symbol
    lambda.symbol = new LambdaSymbol((FunType)lambda.type, lambda.lambdaScope, lambda.pos);
    ctx.declare(lambda.symbol);
}
```

其中，MissingReturnError 在推断返回值类型不为 null 二系统维护的返回标志 returns 为 false 时发出，返回值类型存在其他错误的情况下报错 IncompatRetTypeError。

IV.在所有可能出现 Expr 的地方添加 accept 遍历，包括 visitReturn，visitVarSel 等。

至此完成了该部分功能代码。

C．函数变量及函数调用
a.前期工作:
1°修改 Tree.java 中 Call 的定义。

b.相关工作:
函数变量的引入导致 Call 类型需要对有无 Receiver 进行两套逻辑的梳理，将无函数变量的逻辑留在 visitCall 和 typeCall 中实现（大体同 PA1-A），有函数变量的交给 visitVarsel 实现。

1°分有 Receiver 的两种情况进行讨论

Ⅰ.有 Receiver

```
// 对函数/方法名引用
                if (symbol.get().isMethodSymbol()) {
                    var func=(MethodSymbol)symbol.get();
                    expr.symbol=func;
                    expr.type=func.type;
                    if (func.isMemberFunc()) {
                        expr.isMemberFuncName = true;
                        if (ctx.currentMethod().isStatic() && !func.isS
tatic()) {
                            // static 方法中调用非静态方法
                            issue(new RefNonStaticError(expr.pos, ctx.c
urrentMethod().name, expr.name));
                        } else {
                            expr.setThis();
                        }
                    }
                    return;
                }
```

Ⅱ.无 Receiver

```
if (field.isPresent() && field.get().isMethodSymbol()) {
            var func = (MethodSymbol) field.get();
            if (func.isMemberFunc()) {
                expr.symbol = func;
                expr.type = func.type;
                expr.isMemberFuncName = true;
            }
        }
```

完成以上特性之后，可以通过所有样例点。

四.思考题

1.实验框架中是如何实现根据符号名在作用域中查找该符号的？在符号定义和符号引用时的查找有何不同？

答:

1）本框架本阶段使用的是访问者模式，可以通过 accept 函数打开作用域栈，在栈中从顶向下每一层进行寻找，即可遍历搜索到该符号。

2）符号定义时，会进行“已有声明”的矛盾检查（本质上是在作用域范围内检查命名冲突），通过这个过程找到有冲突的符号。

2.对 AST 的两边遍历分别做了什么事？分别确定了哪些节点的类型？

答:

1） 第一遍遍历（Namer）的时候完成符号表的建立，同时完成了部分类型推导的任务。

第二遍遍历（Typer）的时候主要是在对 AST 中的节点进行类型推断。

3.在遍历 AST 的时候，是如何实现对不同类型的 AST 节点分发相应的处理函数的？请简要分析。

答：在访问者模式下 accept 函数可以打开作用域栈，在需要进行处理的节点对应的节点类中重写基类的 accept 函数，即可通过该函数打开作用于栈，再在该函数中调用不同的处理函数，即可完成相应的处理。