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。

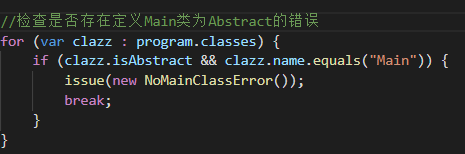
2°在ClassDef类中添加指示类是否为抽象类的参数isAbstract，在建立ClassSymbol类的时候，访问该值并由此判断这个Symbol对应的类是否为抽象类。

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

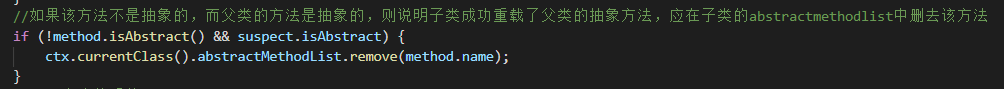
在Namer.java中修改VisitTopLevel函数，添加①类是否名为Main，②类是否为抽象类的判断逻辑。如果某个类被定义为Main且为抽象类，需要报错NoMainClassError。



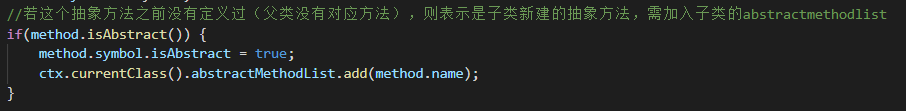
2°含有抽象成员的类不能被声明为非抽象类

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：

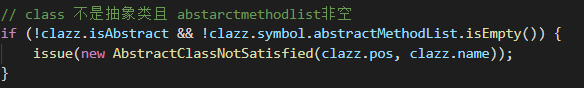
a.删除操作：在VisitMethodDef类中进行修改。在该类判断是否成功重载的函数体内加入，如果成功重载了父类的抽象方法，则将方法名从List中删去的操作。



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。

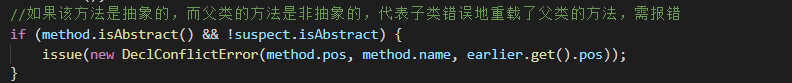


c.遍历操作：在VisitClassDef类中进行修改。如果某个类被声明为非抽象的但它的AbstractMethodList不为空，需要报错。



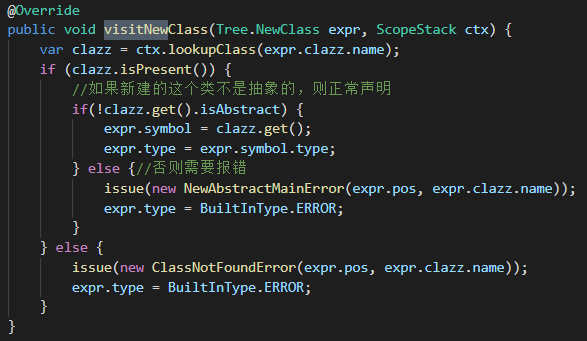
3°已经被定义的非抽象类不能被再次声明为抽象类:

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。



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。



至此完成了ABSTRACT相关部分代码的编写。

2.局部类型推断：

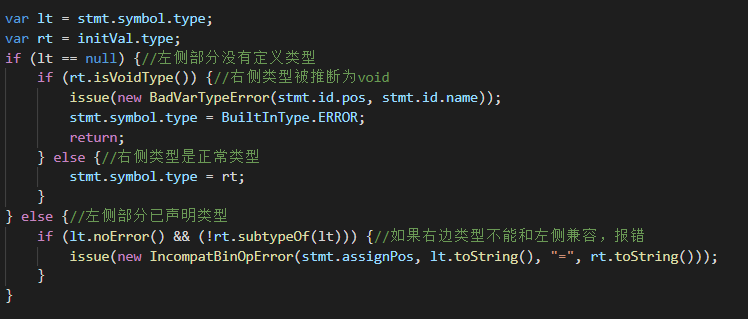
a.前期工作：

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。



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。



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

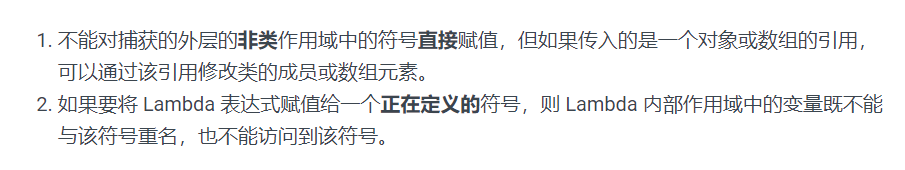
a.前期工作：

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：



据此编写判断逻辑：

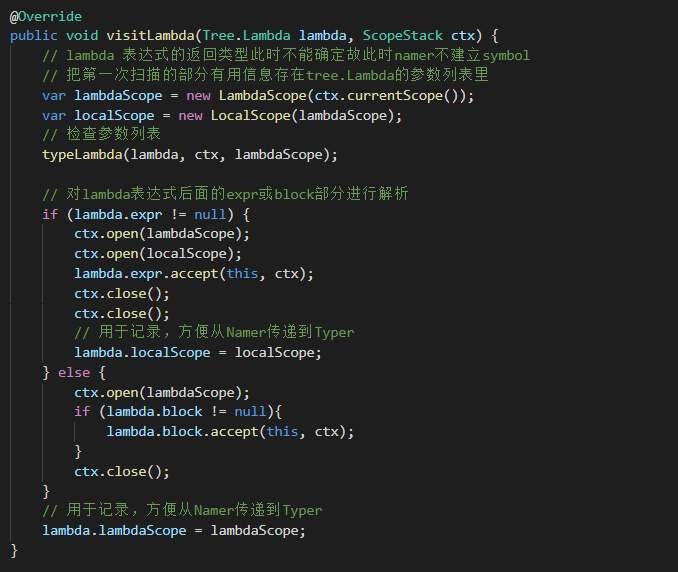


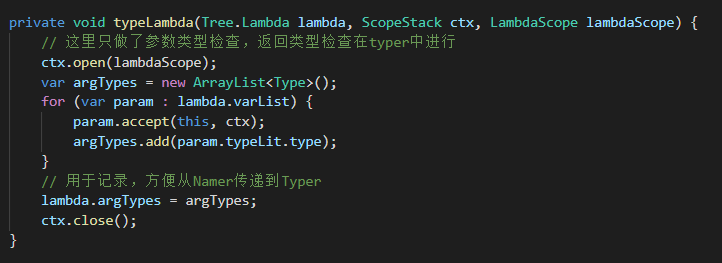
在此处实现了所有不能修改值情况的检验。

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。

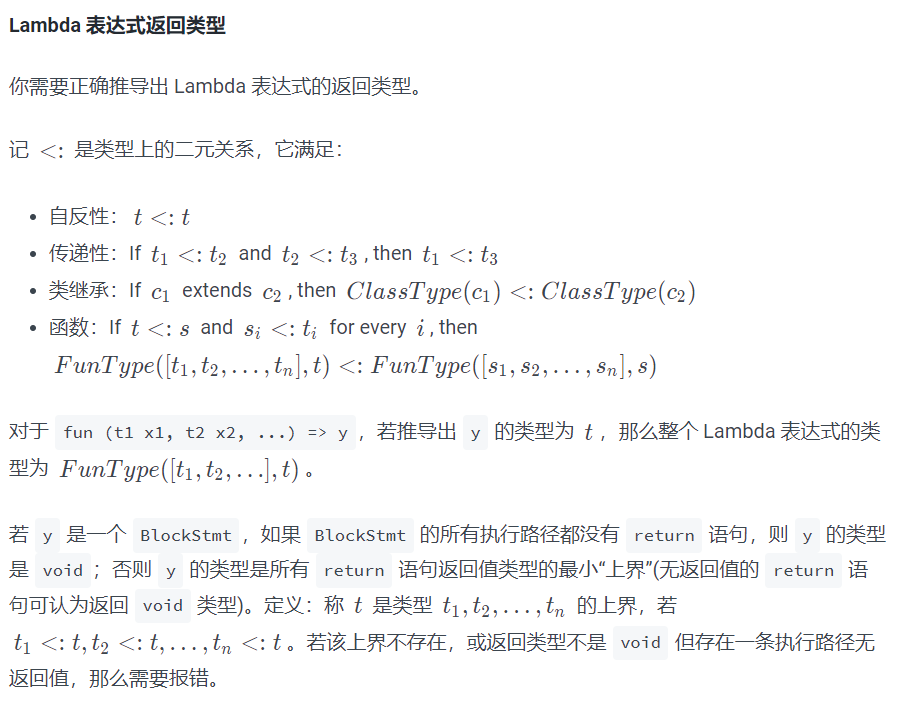
1°Namer.java – VisitLambda类 + TypeLambda类。实现Lambda表达式的遍历，梳理出所有的参数类型并进行类型判断（使用到上面已经完成的部分逻辑）。





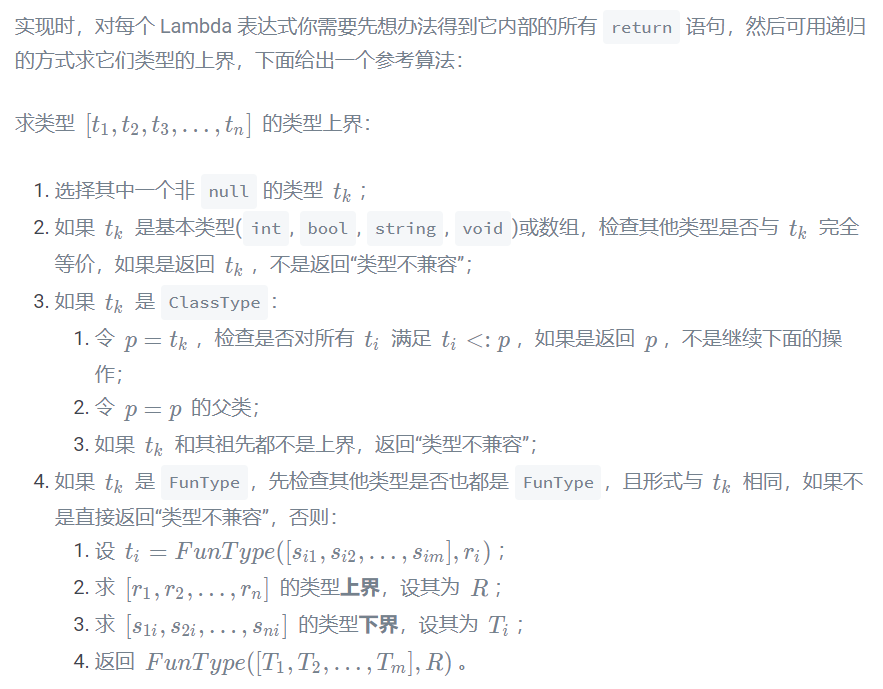
2°返回类型推导：

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：



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。



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.isArrayType()) {

            // 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.isArrayType()) {

            // 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);

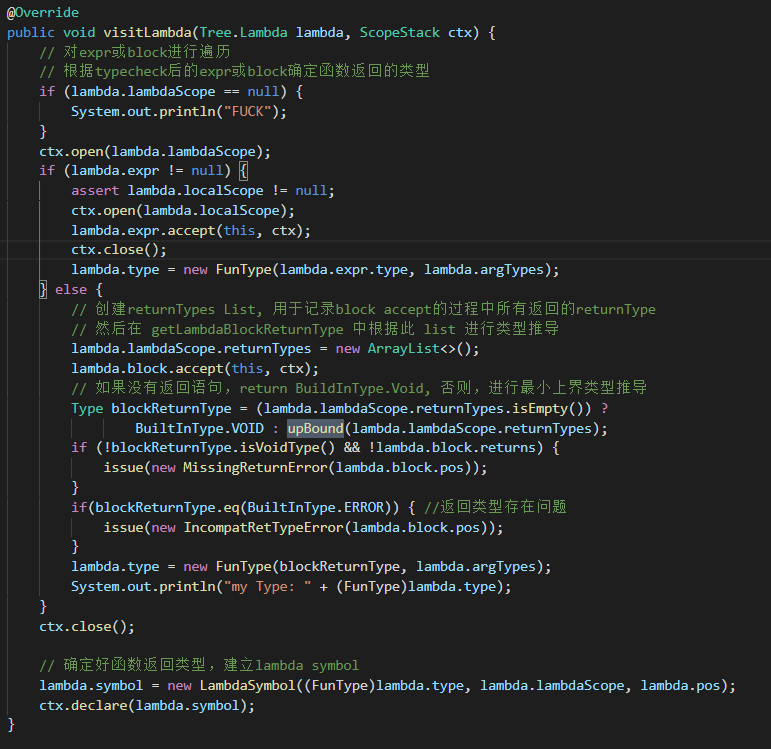
            }

            return new FunType(returnType, retArgList);

        }

    }

Ⅲ.完整的visitLambda代码



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.isStatic()) {

                            // static方法中调用非静态方法

                            issue(new RefNonStaticError(expr.pos, ctx.currentMethod().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。