# Decaf PA2 实验报告

**2017011462 方言**

## 一、新特性：

这部分的`Decaf.jflex`文件沿用PA1-A，`Tokens`和`Parser`中注册新关键字的方法与PA1-A中相同，不再赘述。

### 1、特性一：抽象类

加入`abstract`关键字，可用于修饰类和成员函数。

**实现思路和步骤**

- `Decaf.jacc`中`ClassDef`和`MethodDef`的修改方法与PA1-A几乎一致，在此不赘述。
- 在`ClassSymbol`中加入`Modifier`用来记录其是否为`Abstract`的信息。
- 在`ClassSymbol`中用`List`记录下其未被重载的抽象方法。初始化时与其父类保持一致（若无父类则为空）。在`visitMethodDef`中维护这个列表：当尝试重载一个方法时，若父类中为抽象方法，则从列表中删去该项。反之，若当前为新的抽象方法，则在列表中加入此项。注意这里允许抽象方法重载抽象方法。
- 在`visitClassDef`中，分析结束前判断此列表是否为空，并进行相应报错。
- 同理，在`visitNewClass`中，若实例化一个抽象类，则进行报错。

```
public void visitClassDef(Tree.ClassDef clazz, ScopeStack ctx) {
        if (clazz.resolved) return;

        if (clazz.hasParent()) {
            clazz.superClass.accept(this, ctx);
            clazz.symbol.unoverrideFunc.addAll(clazz.superClass.symbol.unoverrideFunc);
         }

        ctx.open(clazz.symbol.scope);
        for (var field : clazz.fields) {
            field.accept(this, ctx);
        }
        ctx.close();
        clazz.resolved = true;

        if(!clazz.modifiers.isAbstract() && !clazz.symbol.unoverrideFunc.isEmpty())
        {
            issue(new BadAbstractClassError(clazz.pos, clazz.name));
        }
    }
```

### 2、特性二：局部类型推断

加入`var`关键字，用来修饰局部变量

**实现思路和步骤**

- 在`Namer`的`visitLocalVarDef`中，通过判断`typeLit`来确定是否是`var`关键字：

```
if(!def.typeLit.isPresent())
{
    var symbol = new VarSymbol(def.name, null, def.id.pos);
    ctx.declare(symbol);
    def.symbol = symbol;
    if(def.initVal.isPresent())
    {
        var initVal = def.initVal.get();
        initVal.accept(this, ctx);
    }
    return;
}
```

这里我先将生成的`symbol`的类型置为null，这是由于暂时类型尚未确定，应当等到`Typer`中确定了右侧`initVal`表达式的值类型，再对左边的`VarSymbol`的类型做修改。

- 在`Typer`中根据右侧`initVal`的类型做相应修改，并处理相应报错。

```
if (lt == null)
{
    if (rt != null && rt.isVoidType())
    {
        issue(new VoidTypeError(stmt.pos, stmt.id.name));
        stmt.symbol.type = BuiltInType.ERROR;
    }
    else
        stmt.symbol.type = rt;
}
```

## 3、特性三：First-class Functions

### 3.1 函数类型

- 在`TypeLitVisited`中，增加`TLambda`用于处理函数类型（实际上更应该写为`TFunction`），同时在其中处理相应报错。

```
default void visitTLambda(Tree.TLambda that, ScopeStack ctx) {
        boolean flag = true;
        that.returnType.accept(this, ctx);
        var argTypes = new ArrayList<Type>();
        for (var param : that.typeList) {
            param.accept(this, ctx);
            if(param.type.isVoidType()){
                issue(new FunTypeArgsVoidError(param.pos));
                flag = false;
            }
```

```
            else
                argTypes.add(param.type);
        }
        that.type = flag ? new FunType(that.returnType.type, argTypes) :
BuiltInType.Error;
    }
```

### 3.2 Lambda表达式

实现流程基本类似，此处做简要叙述。较为关键的部分为：

- 增加新的符号和作用域（`LambdaSymbol`和`LambdaScope`），分别参照`MethodSymbol`和`FormalScope`实现。
- 在`Namer`中针对`Lambda`表达式的两种类型，分别做处理

```
ctx.open(formal);
var argTypes = new ArrayList<Type>();
for (var param : expr.params) {
    param.accept(this, ctx);
    argTypes.add(param.typeLit.get().type);
}
ctx.close();
expr.type = new FunType(null, argTypes);

ctx.open(scope);
if(expr.body.isPresent())
    expr.body.get().accept(this, ctx);
else
{
    var local = new LocalScope(scope);
    ctx.open(local);
    expr.expr.get().accept(this, ctx);
    ctx.close();
}
ctx.close();
```

  与之前类似，由于无法确定`Lambda`表达式的返回类型，所以先将其设为null，只处理参数类型，返回类型在之后`Typer`中再进行处理。

  注意到`Block`中会单独创建一个`LocalScope`，因此无需再次创建，只需在处理`expr`的情况下新建一个`LocalScope`。

- 判断访问权限。

  我使用了一个栈来记录每一个`LambdaSymbol`，这样可以处理嵌套的`lambda`表达式情况。

  在`visitAssign`中处理

```
if (ctx.currentScope().isLocalScope() && ctx.currentMethod().isLambdaSymbol())
{
    if(stmt.lhs instanceof VarSel)
    {
        VarSel lhs = (VarSel)stmt.lhs;
```

```
        var lambda = (LambdaSymbol)ctx.currentMethod();
        var earlier = ctx.lookupBefore(lhs.name, lambda.pos);
        if(earlier.isPresent() && lambda.scope.get(lhs.name) == null)
        {
            if(lhs.symbol.isVarSymbol() && !((VarSymbol)lhs.symbol).isMemberVar())
            {
                issue(new LambdaAssignError(stmt.pos));
            }
        }
    }
}
```

若当前所处在某个`Lambda`表达式内部，则在该`Lambda`表达式定义位置之前进行查找相应使用的变量，若找到，且判断其不是类作用域中的成员或是数组，则进行报错处理。

- 类型推断
  - `Lambda`表达式刚定义时没有定义`returnType`，因此在`Typer`需要进行返回类型的判断。
  - 若`Lambda`表达式为简单的`expr`形式，则只需分析`expr`获取其类型。
  - 若`Lambda`表达式为带有`Block`，则需要根据提供的算法计算出其返回类型。（在这一部分中需要修改`visitReturn`，在`LambdaSymbol`中记录该表达式所有的返回语句的类型，同时暂时跳过`return`类型不匹配的报错）

```
ctx.open(formal);
if(lambda.expr.isPresent())
{
    lambda.expr.get().accept(this, ctx);
    lambda.symbol.type.returnType = lambda.expr.get().type;
}
else
{
    lambda.body.get().accept(this, ctx);
    lambda.symbol.type.returnType = calcTypeBound(lambda.symbol.ret_types, true);
}
ctx.close();
lambda.type = lambda.symbol.type;
```

### 3.3 函数变量和函数调用

这两部分实现较为复杂，涉及到的修改很多，因此这里不详细介绍，只大致说明思路。

- 原先的`Call`节点分析过程主要处理对成员方法的调用，而现在要将其作为变量使用，就需要将其引入到`VarSel`节点的分析之中
- 在`VarSel`中的做法与之前类似，主要检查其访问权限的问题。这一部分代码都在`typeCall`中，只需要移动过来即可，报错类型也与之前几乎相同。
- 注意到这里需要判断成员函数不可以作为左值参与赋值操作，需要相应报错。
- 另外需要特判一下`Array.length`也是一个符合要求的函数变量。
- 统一地，分析为函数变量后相应的`type`都设为`FunType`

```
if (rt.isArrayType() && expr.name.equals("length")) { // Special case: array.length()
    expr.type = new FunType(BuiltInType.INT, new ArrayList<>());
    return;
}
```

- 现在的`Call`节点的分析只需先分析其左侧的`Expr`，若其类型不为`FunType`，则相应报错`NotCallable`
- 正常情况则继续分析其参数个数和类型，进行相应的报错

```
if (funtype.arity() != args.size()) {
    if(!lambdaexpr)
        issue(new BadArgCountError(expr.pos, name, funtype.arity(), args.size()));
    else
        issue(new LambdaBadArgsCountError(expr.pos, funtype.arity(), args.size()));
}
var iter1 = funtype.argTypes.iterator();
var iter2 = expr.args.iterator();
for (int i = 1; iter1.hasNext() && iter2.hasNext(); i++) {
    Type t1 = iter1.next();
    Tree.Expr e = iter2.next();
    Type t2 = e.type;
    if (t2.noError() && !t2.subtypeOf(t1)) {
        issue(new BadArgTypeError(e.pos, i, t2.toString(), t1.toString()));
    }
}
```

- 其余操作即根据测例和标准输出进行调试。

## 二. 问题

### Q1：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：框架中使用一个栈记录了当前开启的所有作用域。查找时先在当前作用域中查找，若找不到，则向前跳转作用域，直到找到该符号（或者跳到global作用域时停止）。

符号定义时使用`findConflict`，若当前处在`LocalScope`或者`FormalScope`中，则其不能与这个`Scope`之前定义的符号产生冲突。若当前处在`ClassScope`或者`GlobalScope`中，则其不能与任何之前定义的符号产生冲突，此时就是在`findWhile`中的判定条件为任意。

符号引用时使用`lookUpBefore`，需要确定当前引用的位置，查找时需要在此位置之前查找（即只能引用一个在之前定义的符号，而不能引用一个在后续才定义的符号）。

### Q2：对 AST 的两趟遍历分别做了什么事？分别确定了哪些节点的类型？

答：第一遍遍历建立符号表，对于一些基本情况进行报错，比如变量重名等等。第二遍遍历进行类型推断和检查，比如推断`Lambda`表达式的返回类型等等。

第一次遍历时主要确定的是一些基本类型节点，如`Int`,`Bool`,`Void`，还有成员方法的类型等等，但是对于`Lambda`表达式，和`var`关键字定义的节点的类型无法确定。

第二次遍历时主要确定`Lambda`表达式的类型，`var`关键字的类型，以及`Call`节点的类型。

## Q3：在遍历 AST 时，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：采用了访问者模式。对每一个节点，都实现了`accept`方法，在具体实现节点时，`accept`中调用了`Visitor`中定义的不同接口，而具体实现这些接口时完成相应的处理函数。使用时，直接调用某个节点的`accept`，根据动态分派机制，会根据这个节点的具体类型，调用相应的处理函数。