声明：参考了徐明宽同学的思路，但并未抄袭代码。

工作内容

1、abstract

在MethodSymbol和ClassSymbol中增加isAbstract()方法，并在ClassSymbol、Tree、Namer中做相应的适配。

主类不能为Abstract：在Namer.visitTopLevel中修改相应逻辑。

抽象方法没有Block（body）：

1. 在FormalScope里将nested添加Optional属性
2. 在Typer.visitMethodDef和Namer.visitMethodDef中修改相应逻辑，当方法为抽象方法时，不进行相关处理，使Namer中传递的nested为Optional.empty()
3. 在PrettyScope中修改相关内容，使得没有block时不进行输出

若一个类中含有抽象成员，或者它继承了一个抽象类但没有重写所有的抽象方法，那么该类必须声明为抽象类（错误1）：

1. 在ClassDef中定义List<Method> notOverrideMethods；
2. 根据Decaf语法规范，“子类能够覆盖继承而来的非静态方法（通过重新定义这个方法实现），但是新的版本的返回类型和参数类型必须和原有方法匹配。”，需要判断满足如下几点，写为新函数isOverride:
    1. 名称相同
    2. a的返回值是b的返回值的subtype
    3. a的参数数量与b的相同
    4. b的每个参数都是对应的a的参数的subtype
3. 非抽象类不能重写抽象类：在Namer.visitMethodDef中修改相应逻辑

不能new一个抽象类（错误2）：

1. 添加NewAbstractClassError类
2. 在Typer.visitNewClass类中增加判断

2、局部类型推导

在Namer.visitLocalVarDef中增加判断，如果变量类型为null（即是由'var'定义的变量），则定义symbol的type为null；在Typer.visitLocalVarDef中增加判断，如果左值的类型为null，则将右值的类型赋值给左值（如果右值的类型为void则报错），并将Symbol.type的final属性去掉以便赋值。

3、First-class Functions

1、函数调用

仿照isClassName，在Tree.VarSel中加入isMethodName并在Typer.visitVarSel做出相应修改，并且对“静态方法中调用非静态方法”进行判断和报错。在Tree.call中删去冗余的methodname，添加NotCallableType错误

2、函数类型

仿照其他方法，增加TypeLitVisited.visitTLambda方法，以及报错BadTLambdaArgError。根据测例，需要接受所有参数类型后再报错并将type设为error（而非中途直接报错并返回）

略微修改visitLocalVarDef的逻辑，使其可以接受函数类型

3、Lambda表达式

作用域

参考LocalScope, MethodSymbol实现LambdaScope, LambdaSymbol，增加isLambdaScope等相应的判断

仿照visitMethodDef实现visitLambda，在Namer中修改实现一堆visitXXX。大幅修改了Typer.visitVarSel、Typer/Namer.visitLocalVarDef的逻辑，使其支持Lambda。

返回类型

增加BuiltInType CONF表示两类型有冲突，无法兼容。实现Type.upper、Type.lower函数来获取上下界。对于可能Stmt返回类型为null的情况，添加一个新函数来更新其返回类型，并在Typer.visitXXX中调用。

杂项

修改了许多地方的bug

修改了PrettyScope.pretty使其可以打印Lambda Scope

问题

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?

在Java框架中，查找符号的逻辑写在ScopeStack中，主要为lookupbefore()和findConflict()，它们调用了lookup()、findWhile()。findWhile从内向外遍历作用域栈（参数2）中的符号，判断是否有满足条件（参数3）的指定名称（参数1）的符号。在一个作用域中查找直接对TreeMap搜索指定名称，并判断其是否满足条件。有多个满足要求时，返回最内层的。

在符号定义中，调用findConflict()方法来检查符号定义冲突。调用函数为findWhile(key, Scope::isFormalOrLocalScope, whatever -> true).or(() -> global.find(key))，即要求在参数作用域和是局部作用域中查找是否有同名符号，对符号没有限制。

在符号引用中，调用lookupBefore()方法来检查被引用的符号名是否存在。调用函数为findWhile(key, whatever -> true, s -> !(s.domain().isLocalScope() && s.pos.compareTo(pos) >= 0) && (s.type != null))，在当前作用域之前的**所有作用域**中寻找，在当前定义变量之前的符号（这样可以避免找到当前符号）

2、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?

第一趟遍历（Namer.java），标识符声明冲突、非法定义void变量、生成作用域与标识符表，并且生成基础的局部变量与类成员变量类型（TInt、TBool、TArray、TVoid、TString、TLambda以及无需局部类型推导的LocalVarDef、MethodDef、ClassDef）。

第二趟遍历（Typer.java），根据Namer中已经得到的部分类型推断其他所有类型，并进行类型检查、建立标识符与声明的对应、判断语句是否正确。

3、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。

采用访问者模式，设计接口Visitor<C>：

```
public interface Visitor<C> {
    default void visitTopLevel(Tree.TopLevel that, C ctx) {
        visitOthers(that, ctx);
    }
    ...
}
```

并在树的节点中，重写TreeNode的方法：

```
public abstract class TreeNode implements Iterable<Object> {
    ...
    public abstract <C> void accept(Visitor<C> v, C ctx);
}
public static class TopLevel extends TreeNode {
    ...
    @Override
    public <C> void accept(Visitor<C> v, C ctx) {
        v.visitTopLevel(this, ctx);
    }
}
...
```

这样，在调用accept函数之后，会自动转发到Namer或Typer对应的visitXXX方法中，因此仅需在Namer/Typer中实现visitXXX方法，并在需要处理该节点的位置调用XXX.accept(this, ctx)即可。