编译原理 PA2 实验报告

郑逢时 2016012177

2019年11月13日

1. 完成过程

本次PA无论是工作量、算法难度还是对框架语言的理解难度，都比起前两次PA有过之而无不及。四天四夜，多少个腰酸背疼的劳作，多少个心力交瘁的午夜，多少挥"发"如雨的时光，我都这无形的"恶龙"面前度过了。所幸一路不忘初心，凭借着兴趣、毅力、DDL的驱使，在助教的帮助下，我终于完成了本次PA，就像编原书上那勇往直前的"屠龙骑士"一样。当看到满屏幕鲜绿色的 pass 时，我老泪纵横、喜极而泣，那接近三位数的挑灯夜战的小时终于没有白费，（希望）我的付出是值得的吧！

~~感谢你看我废话子这么多。~~下面进入正题。

1.1 abstract 特性

1.1.1 拒绝没有完全实现抽象方法的具体类

实现方法完全参考实验文档的提示，即对每个 ClassDef，维护一个 unimplemented methods 的表

```
pub unimpl_mthds: RefCell<HashSet<&'a str>> // unimplemented methods
```

在symbol pass中，每当开始扫描一个 class 的时候，就将其父类的 unimpl_mthds 复制给自己

```
c.unimpl_mthds.borrow_mut().clone_from(&p.unimpl_mthds.borrow()); // init
unimplemented methods from base
```

之后在 func_def 的扫描中，一旦发现一个抽象函数，就在其所属的类的 unimpl_mthds 中添加这个函数名，一旦成功定义了一个非抽象函数，就从该函数所属的类的 unimpl_mthds 中划去这个函数。

最后在扫描完这个类后，检查其是否抽象并且实现了所有抽象方法。

1.1.2 不能实例化抽象类

在 type pass 的某个 NewClass 分支中拒绝掉对抽象类的实例化即可。

1.1.3 输出 ABSTRACT

为通过测例，我们需要对抽象函数和抽象类输出 ABSTRACT 符号。

具体就是在 Symbol 的 debug print 方法中修改对函数和类的输出，当它是抽象的时候，先输出 ABSTRACT

1.1.4 其他细节

基础框架已经对不规范的 Main class 做了检查，但现在引入 abstract 关键字，需要修改检查的细节，拒绝定义 abstract Main。

在检查成员函数重载的时候，现在新增了一种冲突情况：不能将父类的具体函数重载为抽象函数（反之是可以的）。

1.2 var 特性

本地类型推导的实现较容易。容易想到，symbol pass阶段由于不检查本地变量定义的初始值类型，所以 var 类型也无法推导出来，因此在此阶段我们只定义变量符号，不确定它的type。在 type pass 阶段，我们再对 var 类型确定其 type，直接取右端初始值的 type 即可。

有一个特殊情况，就是右端初始值的类型如果是 void，需要报 VoidVar 错误（例如 `var a = print(...);` ）

1.3 First-class Functions

如同文档所说，这是到目前为止我遇到的最棘手、最复杂的一个特性。每个小特性的实现都举步维艰，其中的困难部分移至下一章节讲解。

1.3.1 函数类型参数

首先，我们在上一次PA中完成了对语法树上函数类型 SynTy::FunType 的定义和debug输出，现在我们要将其转换为 Ty 以便做类型检查。转换过程可以可以递归地进行，collect 成一个 vector，再用转换为数组引用存起来。这个转换比较 tricky，在 2.1 中介绍。

注意在转换的过程中同时检查有没有出现 void parameter，若有需要汇报错误。

1.3.2 扫描 lambda 表达式作用域

基础框架只对 FuncDef 中的 local scope 扫描，构建出对应的符号表。现在新引入了 lambda 表达式，新的规范要求我们首先构建其参数作用域，再构建主体作用域。注意到其主体可能为 expr，也可能为 block，block 自然有可能嵌套又一个 lambda 表达式，而 expr 也可能返回一个新的 lambda 表达式，因此我们扫描构建 lambda 表达式作用域的过程必须**递归地**进行。

首先迎来的一大挑战是，symbol pass的基础实现并没有通篇扫描AST，只是提取其中的 VarDef 等定义信息。现在我们必须遍历整个 AST，不放过可能藏在犄角旮旯的任何一个 lambda 表达式。解决方案详见 2.2 节。

遍历到 lambda 表达式后，需要将其符号存入当前作用域，于是我们要引入新的 Symbol —— `Symbol::Lambda(&'a Lambda<'a>')`，其内含一个 AST 中 Lambda 节点的引用，并在 symbol.rs 中补全其相应的 helper 方法。

其次，我们需要打开一层新的作用域，深入扫描 lambda 的参数列表和主体。为了后期方便，我又新引入了新的 ScopeOwner —— `ScopeOwner::LambdaParam(&'a Lambda<'a>)`，在 symbol.rs 中补全其 helper 方法。（当然为了区分成员函数的参数域，我特意将 Param 更名为 FuncParam）

打开了新的 lambda param 作用域后，扫描所有变量构建形参符号表，之后新的问题接踵而至。

紧接着我们要打开 lambda 主体的 local scope，但是因为 lambda 表达式主体可以是 expr 也可以是 block，我们必须分情况讨论：

- 如果是 block，那么很简单，直接打开 block 的作用域，复用已有框架代码扫描之；
- 如果是 expr，问题来了，Expr 这个结构并不自带 scope 字段，这个只有 Block 结构有，现场临时造一个 Block 又会导致 borrow checker 不通过。
    - 折衷考虑之下我选择了重构 `LambdaKind::Expr` 这一枚举类型（详见我的第一次PA报告），让其成为二元组，第一个位置存 Expr 节点，第二个位置用来存它的 Scope。

```
pub enum LambdaKind<'a> {
    Block(Box<Block<'a>>),
    Expr(Box<Expr<'a>>, RefCell<Scope<'a>>), // if this is a expr-lambda, it
should contain a local scope
}
```

    - 但是 `ScopeOwner::Local(&'a Block<'a>)` 只能接受 Block，不接受 Expr。我在这个问题上来回折腾了好长时间，最后选择了一种不太优雅但是可行，并且几乎是 zero overhead 的方案：将其重构为 `Local(Option<&'a Block<'a>>, &'a RefCell<Scope<'a>>)`，这样如果是 block 就让元组的第一个位置填 Some(block)，第二个取 block.scope；如果是 expr 就让第一个位置空着（None），第二个取 `LambdaKind::Expr` 附带的第二个位置的 scope 的引用。

另一个问题是，后续的遍历中会经常问，当前的作用域是否在一个 lambda 表达式里面。为此我在 scope_stack.rs 中新增了一个 helper，用来随时查找当前作用域栈中最顶层的 LambdaParam scope：

```
// look for the uppermost lambda param scope
pub fn cur_lambda(&self) -> Option<&'a Lambda<'a>> {
    self.stack.iter().rev()
    .filter_map(|&owner| if let ScopeOwner::LambdaParam(l) = owner { Some(l) }
else { None })
    .next() // yields the first hit
}
```

有了这些新结构和 helper functions 后，我们就可以比较顺畅的遍历寻找 lambda scope 了。

1.3.3 推导 lambda 表达式返回值类型

2.2 中提到，我基本复用了基础框架 type pass 中扫描 block、stmt 的代码来扫描 lambda scope，但是相应的代价是要在运行时增加对当前是否处于 lambda scope 的判断开销，以完成特殊的语法动作。

例如，在推导 lambda 返回类型之前我们需先收集所有的 return 表达式的返回类型（这部分算法完全参考实验文档）。那么我们在 `StmtKind::Return(r)` 中判断当前是否处于 lambda scope，若是的话，直接将返回类型收集到 Lambda 节点里面。为此，我又在 Lambda AST节点上新增了一个 can_tys 字段，是一个 Ty 的 Vec。收集 return types 的相关代码：

```
StmtKind::Return(r) => {
                let actual = r.as_ref().map(|e|
self.expr(e).0).unwrap_or(Ty::void());
                match self.scopes.cur_lambda() {
                    None => {  // in normal method body
                        ...
                    }
                    Some(l) =>  // in a lambda's body, we gather candidate
return ty here
                        l.can_tys.borrow_mut().push(actual)
                }
                actual != Ty::void()
            }
```

接着，个人认为是本次PA最难的部分来了——从多个返回值种类归纳出**最小上界**。好在文档给出了一种可行的算法，我们只用仔细实现，故不列入“所遇困难及解决方案”。

首先文档已经定义了什么是“可赋值的”（assignable），可以看成是一种 Ty 上的**偏序关系**。很多个类型放在一起构成了一个偏序关系图，问题即求出这个关系图的“上确界”，如果有的话，否则报错。

算法的基本思想是 reduce and conquer，即“减而治之”，递归求解。具体内容这里不细说了，只说一些实现细节。具体实现在 ty.rs 的 `Ty::sup` 和 `Ty::inf` 这一对函数中。传入都是 `(tys: &Vec<Ty<'a>>, allocator: &'a Arena<Ty<'a>>)` 返回 `Result<Ty<'a>, FailToDetermineTy>`

在实际实现时，先判断传入的 tys 是否为空，空就直接返回了。然后 Ty 中有种叫 error 的类型，这是在语义分析过程中出错时的占位符，它不应该影响对上确界的求解，所以可视作所有类型的子类型。为此，我的程序先扫描了一趟 tys，在求上确界的时候，直接过滤掉所有的 error ty，因为他们不影响分析；在求下确界的时候，发现了 error 就直接返回 `Ok(Ty::error())`。然后再扫一遍，判断数组维度是否一致。最后：

- 求上确界时，按照文章描述算法，先过滤掉所有的null类型得到新的数组，讨论其第一个元素：
  - 如果是基本类型，除了要检查其他都是否一模一样外，还要看之前过滤null时有没有出现null，如果有则求解失败；（因为null并不是基本类型的子类）
  - 如果是类类型，……

    - 如果是函数类型，同样先看之前有没有出现null，若有直接失败，没有再按照算法流程递归求解…… 最后求出来的结果还要转化为 `TyKind::Func(&[Ty])` 数组，使用 2.1 节描述的那种技巧，通过调用 sup 时传入的 allocator 解决；
    - 如果没有元素了，讨论有没有error和有没有null来返回适当值。
- 求下确界时，讨论 tys 第一个元素：
    - 如果是基本类型，处理与上确界相同……
    - **如果是类类型**，先确保剩下的元素都是类类型，然后判断这几个类是不是都在**同一条继承链**上，若是则返回最子类的那个，否则失败；（大概的证明：因为decaf只支持单继承，所以所有的类继承关系可以画成一棵树或者森林，父是基类，子是子类；如果 tys 是可兼容的类，假设 tk 是他们的下确界，则对任意 tys 中的 ti 和 tj，由 tk <: ti, tk <: tj 及 tk 是单继承的事实可以推出要么 ti <: tj 要么 tj <: ti。这样就证明了 tys 中的任意两个 ty 都是继承关系，自身构成一条继承链）
    - 如果是函数类型，……

1.3.4 lambda 作用域中的访问权限控制

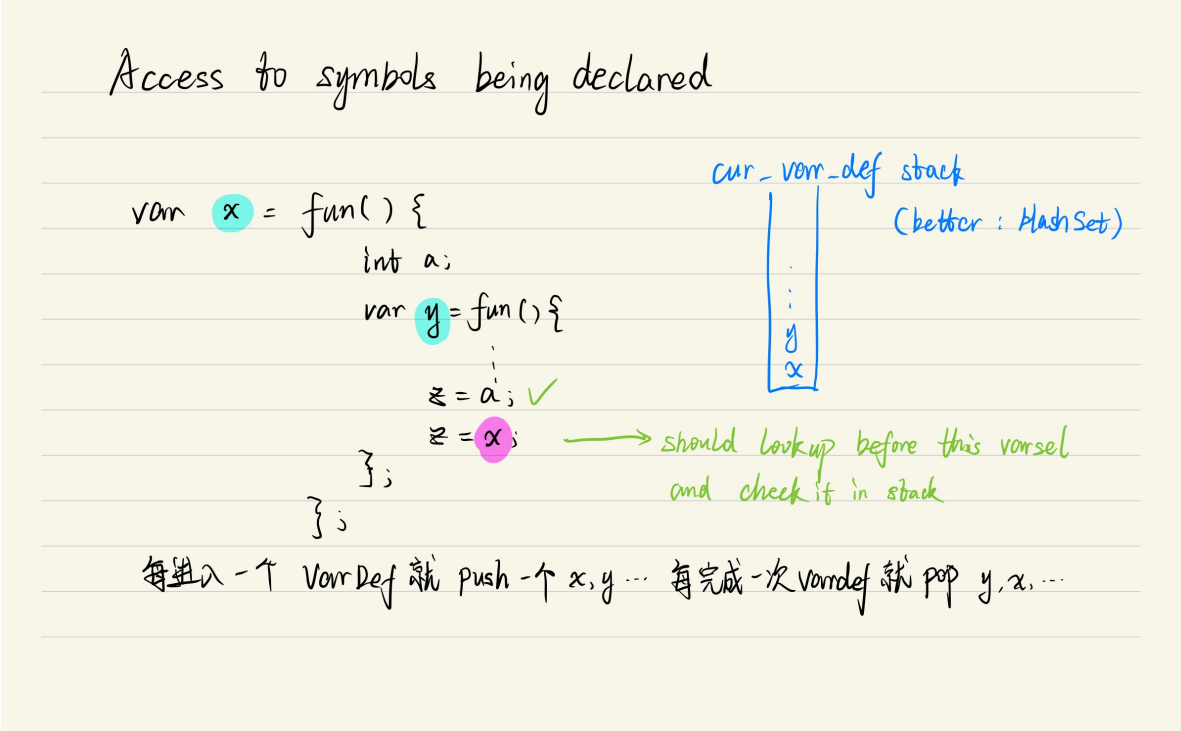
按照文档要求，我们需要拒绝一部分在 lambda scope 中的赋值语句。

对于正在定义的符号，由于现在有嵌套 lambda 定义的情况，正在定义的符号可能有任意多个。为了维护这些符号，我将 `TypeCk::cur_var_def` 改为了栈来维护：

- 每进入一个 LocalVarDef，就在栈中压入一个 var 的名字；
- 每退出一个 LocalVarDef，就弹出一个符号；
- 每当在 var_sel 查询一个需要访问的变量符号（成员函数符号则不存在这个问题），在符号前面的scope中查找找到了symbol，就在栈中看这个symbol是否已经存在，若是则报 UndeclareVar 错误。

实际实现后，将栈的数据结构从向量换成了哈希集合，希望性能更优越。

道理如下图所示，蓝色高亮的符号正在被定义，已加入栈中，试图访问红色高亮的符号就会出错

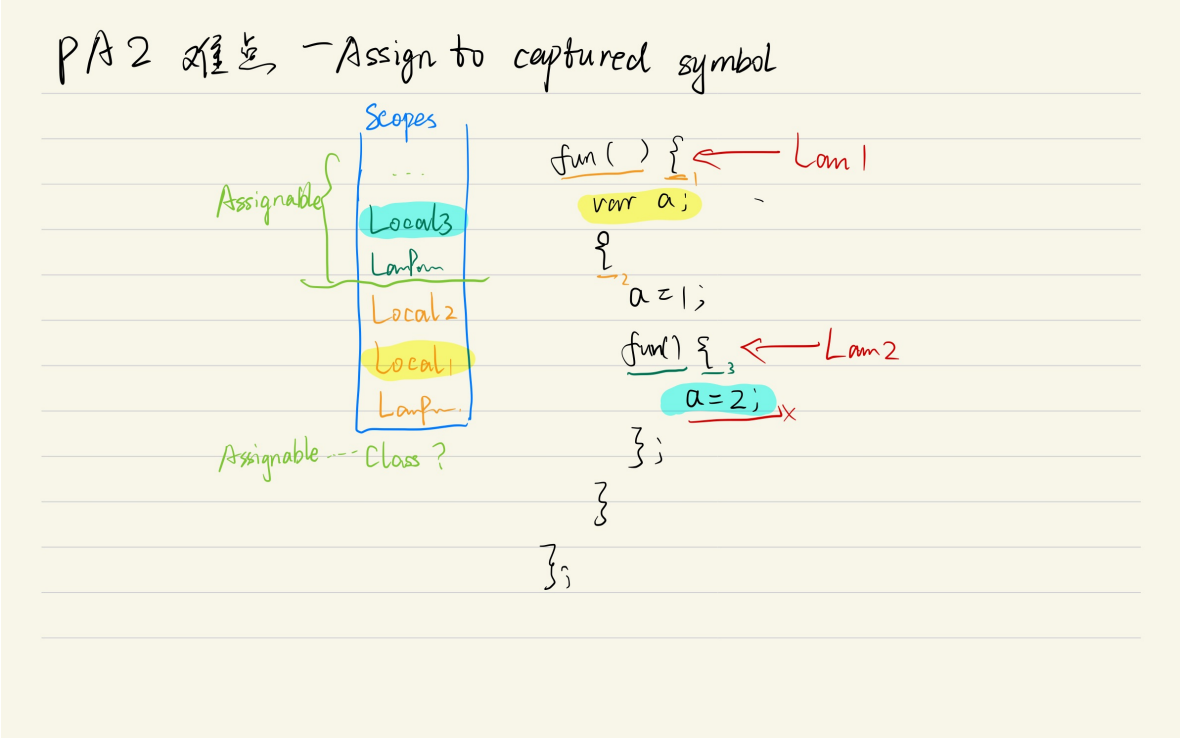


对于赋值语句的权限检查，现在我们要拒绝在某个 lambda scope 中对外层非类成员变量赋值的情况。当然，如果试图给一个成员函数符号赋值，要报 AssignToMemberMethod 的新错误。这些规范的实现难度也较大，主要是需要给出 assignment 左值的符号名和所属的作用域，才能进一步判断，这部分 trick 放在 2.3 节。

假如我们现在可以得到左值捕获到的符号名和所属的作用域了，现在讨论当前语句所属的 scope：

- 如果不被任何 LambdaParam scope 包含（这个可以用我们前面实现的 cur_lambda 这个 helper），则按照基础框架处理……
- 如果在一个 lambda 的 scope 里：
  - 如果捕获到是一个 Class scope 中的函数符号，报 AssignToMemberMethod 错；
  - 如果是一个 Class scope 中的变量符号，可以赋值，pass；
  - 其他情况，如果访问到的位置比 cur_lambda 的位置更早，说明访问到外层非类作用域符号了，应拒绝赋值，报错。

道理如下图所示，蓝色高亮的 Local3 是当前所在scope，它能访问到的只有绿色的两个 scope，然而却试图访问一个位置比它靠前的外层 Lambda scope 里面的符号。



1.3.5 输出 lambda scope

现在输出作用域也需要遍历整棵 AST 里所有可能出现的 lambda expr ，做法和 2.2 节类似，并新增对 lambda 表达式作用域的输出规则：分 expr 和 block 两种情况来输出内作用域即可。

1.3.6 函数调用

由于函数现在是 First-class 的，报错的位置和检查逻辑都要发生变化。粗略地说，是把之前在 call 中处理函数调用的代码转移到 var_sel 中，相当于可以直接访问到一个函数指针，而 call 的处理变得异常简单——只需先检查 caller 部分的expr，看它是否是 callable 的（成员函数指针、函数类型或是数组的 length 方法），最后检查参数和函数签名是否匹配（check_arg_param 方法）。为了保存当前 caller，我在 `TypeCk` 中新增成员 `cur_caller`。

这里有一个和 undeclare var 类似的问题，就是函数调用可以是一重接一重的，例如 `f(1)(2)(3)` 这样的表达式现在就有可能式合法的。call 完一个表达式后，需检查结果类型是否依旧是函数类型，如果是需要更新 cur_caller ，避免后续的连环call对应不到 cur_call。

2. 所遇困难及解决方案

2.1 如何动态地定义数组以保存 TyKind::Func ?

在 SynTy 转 Ty 和推导 lambda 表达式返回值类型（可能返回函数类型）的时候，需要创建 `&[Ty]` 这种东西，我们都不可避免地会触及 rust 的一个红线——不允许动态地定义长度任意的数组。这是因为数组在 rust 中只能是静态长度的，它的本质只是一片地址连续的存储空间的引用。要动态创建引用，我们就需要引入生命周期，而生命周期又要suffer from borrow checker 的检查……种种 rust 对安全性的保证在这里为我们过编译带来了障碍。迷雾重重，我们难道就没有办法动态创建数组了吗?

幸运的是，这是可以做到的。基础框架为我们做了示范，只需用到 `TypeCk::alloc::Ty` 这个工具。

```
pub struct TypeCkAlloc<'a> {
    pub ty: Arena<Ty<'a>>,
}
```

这本质上是一个 rust 的标准库 Areana，它是一种强有力的内存动态分配工具，它的 alloc_extend 方法可以为我们所用，只需接受一个动态的迭代器，通过迭代器迭代过程一边自动分配内存，一边把创建好的连续内存地址以 `&[Ty]` 的形式返回给我们，而这正好是我们的需求。而且 alloc 分配的数组引用的生命周期恰好和 TypeCk 一致，这样就巧妙通过了 borrow checker 的检查。

2.2 如何全方位无死角地扫描和打印所有地 lambda 表达式作用域?

我们的目标是遍历一个 Block 里面的所有 Statements，不放过任何一个语句的任何一个表达式中可能存在的 lambda 表达式。

这听起来太难了，其实只要用递归的思想，不断讨论 stmt、expr 都是些什么类型，然后参考 parser.rs 里面的文法，委派相应的处理函数去处理就行了。事实上基础框架的 type pass 就用到了这种思想。

特别的，为处理地更加细致，尽可能减少运行时的分支判断开销，并提高代码可复用性，我特意在 symbol_pass 中新定义了扫描 simple_stmt, lvalue 语法节点的函数，并将部分 stmt, expr 代码迁移过来。

递归扫描的 base case 落在一个 lambda 表达式的情况分支上：

```
match &expr.kind {
    ...
    ExprKind::Lambda(l) => self.lambda_expr(l),
    ..
}
```

这样就完成了对全树的扫描。

为了节省工作量，在 type pass 中我就没有特意设计这些减少分支的优化了，而是直接复用基础框架扫描 block 和 stmt 的代码来扫描 lambda 作用域。微加入一些对 context 的判断来特别实现 lambda scope 中的一些规范。

2.3 如何给出 assignment 左值的符号名和所属的作用域?

我们希望 type pass 的 expr、var_sel 等方法能顺带地给出这两个信息，以便在 assignment 节点的检查中，一方面能追溯到是哪个符号不能被赋值，报错能输出符号的名字，另一方面是可以追溯 lambda scope 中被捕获符号的作用域来判断赋值权限。

我所采取的方法并不美观，但实现起来比较容易：让 var_sel, expr 能返回一个元组 `(Ty<'a>, Option<(Symbol<'a>, ScopeOwner<'a>)>)`，第一个 Ty 是类型，和基础框架一致；第二个是可选的，对右值表达式或者肯定不会造成 assignment 权限错误的表达式，直接返回 None，对左值表达式和一些比较关键的符号才返回 Some，第一个是捕获到的符号，第二个是所属域。

相应的需要重构一下这两个函数中的代码，难度不大。左值有这样一些情况：

- 访问到一个对象的
  - 成员方法时（如 a.fun），需同时返回符号及作用域，为的是能报错 AssignToMethod；
  - 成员变量时（如 a.x），**不用返回**符号及作用域，因为这样的符号 a 是作为引用存在的，不会造成赋值权限问题（对数组访问的那种分支也是同理）
- 访问到类静态方法，需返回符号作用域；
- 直接访问到一个对象（例如a），必须返回它的符号作用域；
- 直接访问到一个方法（例如fun），也必须返回。

3. PA问题回答

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？

rust框架的作用域是用 `ScopeStack` 这个结构实现的，内含 `global` 和 `stack` 两个成员，前者是全局的Scope，后者是一个Scope栈（本质是Vec），用于动态维护 type check 过程中的开作用域。

实现的查找过程是先在 `stack` 中**反向**迭代每个scope owner，对每个scope（本质为哈希表）再哈希查询符号的名字，若查到即返回符号及相应的作用域，若没有则继续询问下一个scope owner，直到最后对 `global` 的scope做查询。

在遇到符号定义时，先查询被定义的符号在open scope中有没有出现过（特别地，对局部作用域只在当前符号之前查询），若出现过则视具体情况分别报错或者允许（如重载函数是允许的，重载变量则报错OverrideVar），否则没找到或者允许重定义则 declare 这个新的符号到当前scope。

在符号引用时，直接在open scope对当前符号前面查找这个符号，若找到则进一步看这个符号是不是一个Var，是的话还需判断它是否正在被定义（如 `int x = x`, `var f = fun() { f; }` 这些情况），如果是正在定义的一个Var，或者没有找到这个符号，都报Undeclared错误；否则还需看是否在静态域访问类的非静态成员变量、函数等，相应的也要报RefInStatic 错；如果这些都没有发生，才允许访问该符号。总之，访问符号比定义符号的判断分支更为复杂，更为严格。

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？

第一遍：symbol pass，主要是检查类定义的规范性、建立符号和作用域表（哈希表）。

1. 构建Global scope 作用域，建立类节点的符号，检查和保存继承关系，检查有无循环继承等；

2. 检查有没有符合decaf规范的 Main 类及`static void main()`方法；
3. 扫描构建类的作用域：成员变量及成员函数符号。（新特性）检查非abstract类是否完全实现abstract方法；
4. 扫描类成员函数定义，检查有无不规范的重载等，构建Param作用域及参数符号，扫描局部作用域（Block）；
5. 扫描类成员变量定义和局部域语句中的所有变量定义（VarDef），检查有无冲突或不规范的重载等，构建变量符号。
6. （新特性）扫描语句、表达式中的所有 lambda 表达式，构建匿名函数对应的符号、参数符号，递归地扫描 lambda 表达式主体。

这遍扫描中，能确定类型的只有类、成员函数和非局部类型推导的变量定义。

第二遍：type pass，在有了作用域和符号表的基础上更加细致地扫描AST，严格检查和确定所有节点的类型。

1. 对每个类的每个非抽象方法深入扫描，最后检查非空返回类型的方法是否有返回；
2. 对每个block中的所有语句扫描，分各种语句情况分别深入检查，检查有无返回；
3. 检查每个表达式求值后的返回类型，分情况深入；
4. 检查所有 VarSel 的类型，是函数还是变量，返回其类型；
5. （新特性）推导所有本地类型推导 var 的类型；
6. （新特性）推导所有 lambda 表达式的返回类型；
7. 检查 condition 的类型是否为 bool；
8. 检查 return 语句的类型，若在一个成员函数的scope中，检查返回类型是否匹配，若在一个 lambda 主体的scope中，则把当前的 return type 添加到候选列表，最后由 6 确定。

这遍扫描，能完全确定AST所有节点的类型。

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。

rust框架没有采用 visitor 模式，而是在 symbol pass 和 type pass 的处理入口函数 `program` 中面向 AST 遍历过程编程，将不同类型的遍历任务，参照文法规则，分别委托（delegate）给相应的处理函数，如 `block`，`expr` 等。例如：

在 type pass 的 `stmt(&mut self, s: &'a Stmt<'a>) -> bool`函数中，接受参数为当前遍历到的语句在 AST 中的引用。该函数分语句 s 的各种情况：`StmtKind::Assign`，`StmtKind::LocalVarDef`，`StmtKind::ExprEval` 等等。然后每个分支分别规定 type pass 的规范，应该报哪些错，应该委派给哪些函数处理等。

例如对 `StmtKind::Assign(a)` 这种情况，文法是 `Stmt -> LValue '=' Expr ';'`，所以先委托给 `expr` 函数处理 assignment 左右两边的表达式类型（特别地，左边是左值表达式），再判断左右类型能否兼容，最后特殊讨论当前作用域是在一个 lambda scope 中的情况，这时再检查捕获到的变量能否赋值等等。

4. 致谢

特别感谢李晨昊助教对我本次作业的答疑解惑，排除了我对需求的很多误解，没有他的帮助我可能得绕多得多的弯路才能完成本次PA，助教非常的尽职尽责！