**编译PA3：语义分析 报告**

17307130015 王鑫涛

**一、背景知识**

本次PA要实现语义分析，有以下等要求：

检查Main类和Main类中的main函数

main函数不能具有参数

检查类名：

不能命名为基本类名或SELT\_TYPE，不能重复定义。

检查类别继承：

是否循环继承

继承类型是否存在

是否继承了基本类型或SELT\_TYPE

检查类的Feature：

是否重复定义

子类是否重写父类属性

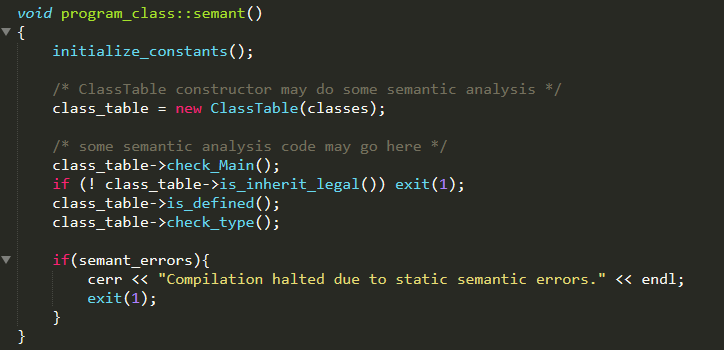
子类重写父类虚方法时Formal和返回值是否一致

检查表达式：表达式返回类型与声明类型是否匹配

在cool-tree.h中，定义了Program、Class、Feature、Formal和Expression以及相关继承子类的类和信息，其中XXX\_EXTRAS部分在cool-tree.handcode.h中定义，我们需要在cool-tree.handcode.h修改XXX\_EXTRAS来完善cool-tree.h的定义。

进入语义分析的接口为semant.cc中的program\_class::semant()函数。我们通过实现这个函数，完成语义分析。为了实现该函数，我们需要实现能完成上述要求的诸多函数并设计相应的数据结构，可以将这些东西实现在semant.cc中。

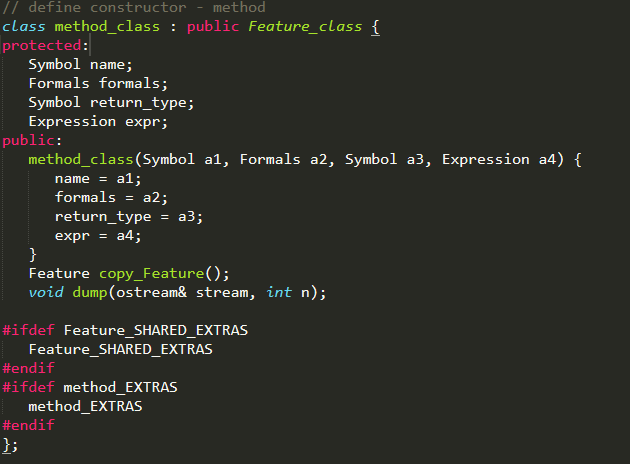
个人的实现将依次检查Main函数是否存在（本报告第九部分）、检查继承关系是否合法（本报告第六部分）、检查各符号是否用良好定义（本报告第七部分）、对各符号进行类型检查（本报告第八部分）。第二部分是头文件中的定义，第三到五部分是相关数据结构的实现。



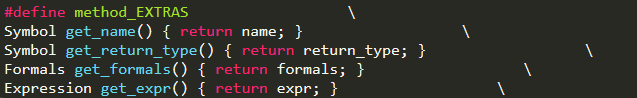
**二、cool-tree.(handcode.)h中的定义**

首先，从cool-tree.h中，我们能看到一些已经定义好的类的结构。以method为例，它具有属性name、formals、return\_type、expr，具有方法dump、构造函数和拷贝函数。

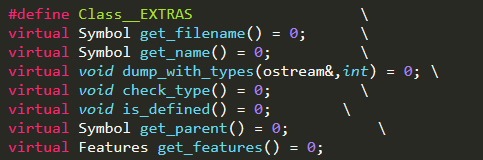
在其下的Feature\_SHARED\_EXTRAS区和method\_EXTRAS区，我们可以定义一些我们想要实现的方法，这一部分的实现应该写在cool-tree.handcode.h中。



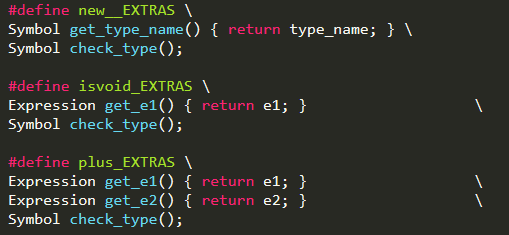
最基础的就是一系列的get函数。因为name等属性是protected的属性，不能直接读取，所以我们要提供public的get接口，来访问这些属性。



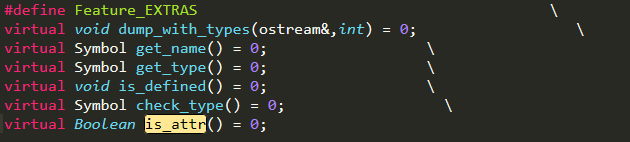
除了一系列get\_xx作为访问属性的接口，我还对很多类定义了check\_type和is\_defined两个方法。check\_type用来检查类型和定义是否合理，这一步往往是递归进行的，所以对于被调用的check\_type函数，可能需要返回其类型以支持上级的check\_type。is\_defined函数用来确认是否被定义。类的check\_type会调用Feature的check\_type，Feature的check\_type会调用formal和expression的check\_type。



cool-tree.h中，为每一种表达式都定义了一个类。每一个类内还提供了divide\_EXTRAS，所以我们还要为这些类完善定义。此处只列举其中几个。

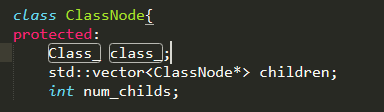


为了区分Feature中的Method和Attr，定义了函数is\_attr。

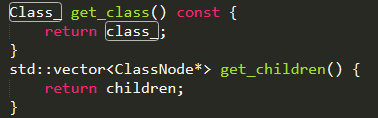


**三、类表节点**

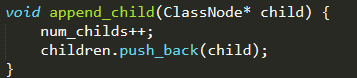
接下来，完善类表的定义。已知类之间的相互继承不能成环，那么继承关系就是一种树状结构，而类是其上的节点。因此，我们以树的结构来组织类。那么，只用cool\_tree.h中的Class\_显然是不够的，我们需要首先将其包装为节点。这个节点需要的属性包括其Class\_、子节点children以及子节点的个数。



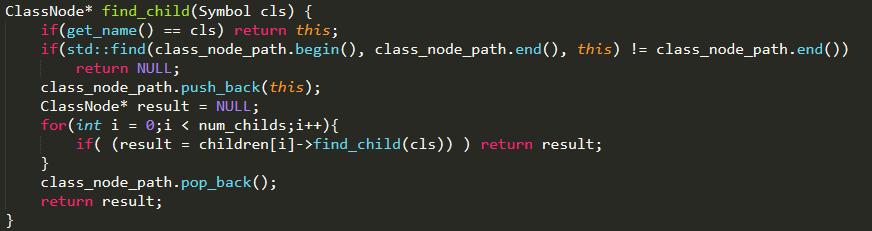
要为其设计能访问私有变量的接口：



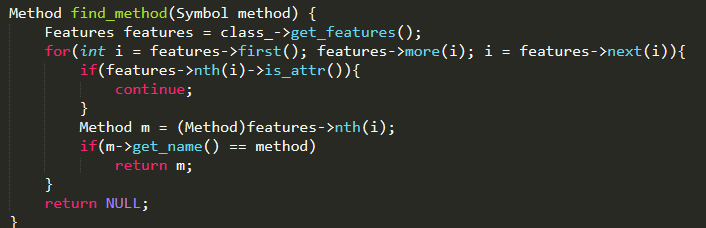
添加子节点（的指针）的方法：



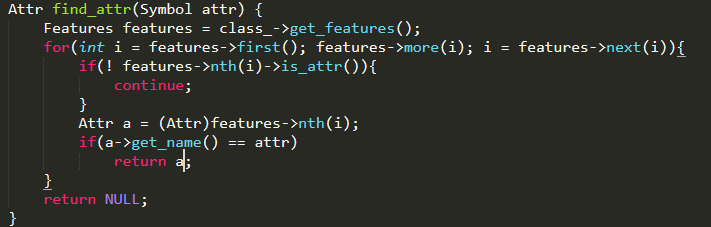
查找子节点时，会从该节点开始，递归地深度优先搜索遍历所有子节点。这个过程中，用一个栈class\_node\_path记录当前路径。如果出现环型结构，则会在第二次经过cls节点时，在栈中找到cls节点，这个时候可以直接返回NULL，就可以避免困在死循环中。



在class上查找方法时，我们只需要找到class对应的features，遍历他们，跳过遇到的attr，比较遇到的method的名字，如果找到就返回。



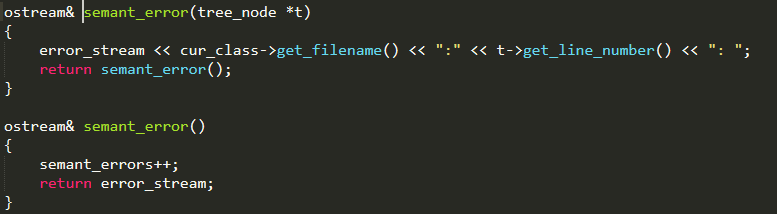
查找属性的函数同理。



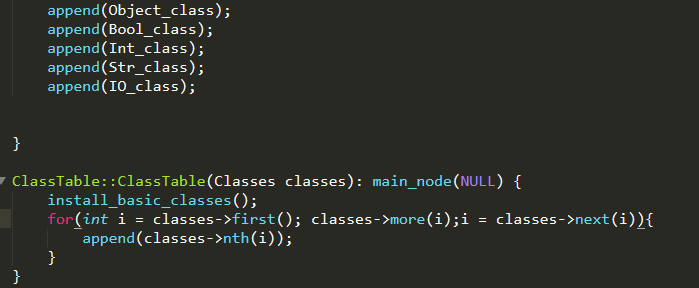
用于递归检查是否定义的函数is\_defined和用于检查类型的函数check\_type将在下面一并介绍。

**四、类表的构造**

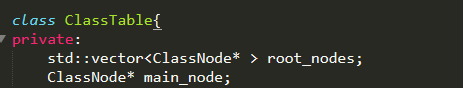
在类表的定义中，首先将错误信息输出函数semant\_error()解耦并重构，以便跟方便的输出错误信息。



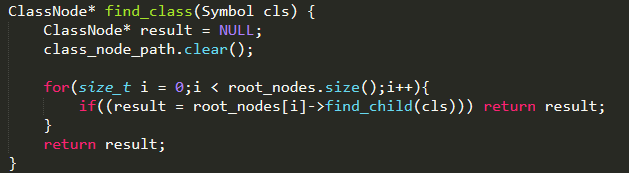
类表的属性只有2个：一个Main类对应的节点，一个是用来存储继承树的根节点的容器root\_nodes。



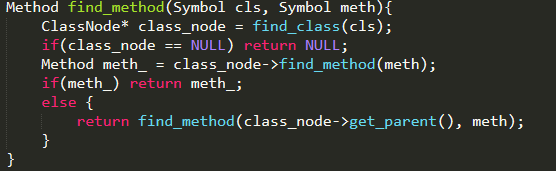
类表的初始化，需要先install\_basic\_classes，我将5个基本类在这个函数中也加入了类表之中。随后，将参数传进来的classes，依次插入的类表中。



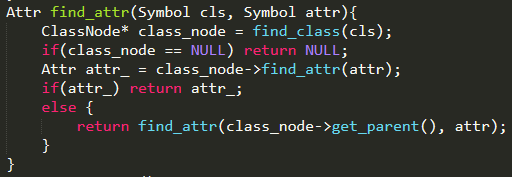
在类表上查找类，是从每个继承树的根节点开始，递归调用find\_child函数。



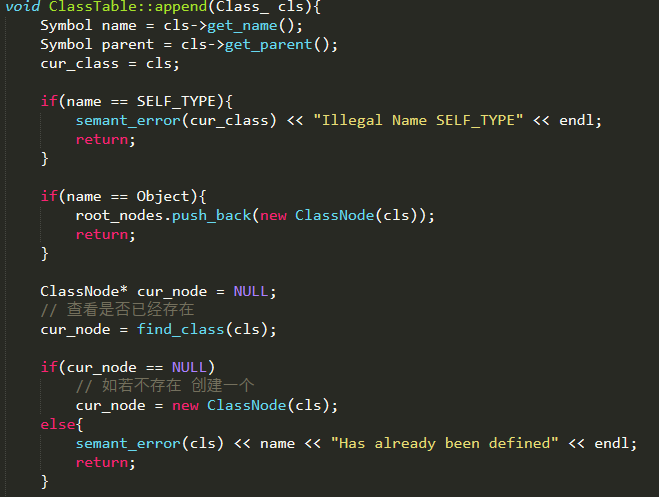
在类表上查找类的方法，首先需要找到这个类，然后看看该类的节点上是否定义了该方法，如果不存在，则递归地对父类进行查找，直到父类为NULL。



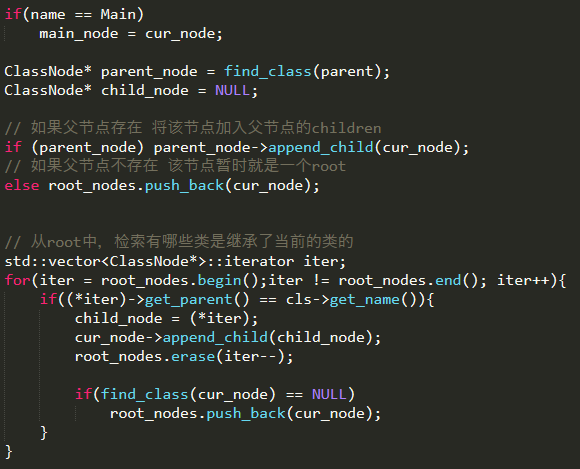
查找属性的方法同理。



类表的插入首先需要对类型进行检查，查看名字是否为SELF\_TYPE，看看类节点是否被定义过，名字是否为Main。



接下来，要检查继承相关信息。如果父节点存在，则将其插入父节点的子节点中。如果父节点不存在，则暂时将自身作为一棵继承树的根节点。然后，去当前的根节点中，检索当前类是否为每个根节点的父节点。然而，如果此时当前节点cur\_node即是child\_node的父节点，又是其子节点，cur\_node已经在child\_node下，删除child\_node则将删除这2个节点形成的环。这种情况应该保留child\_node，留待后面循环继承检测处理。



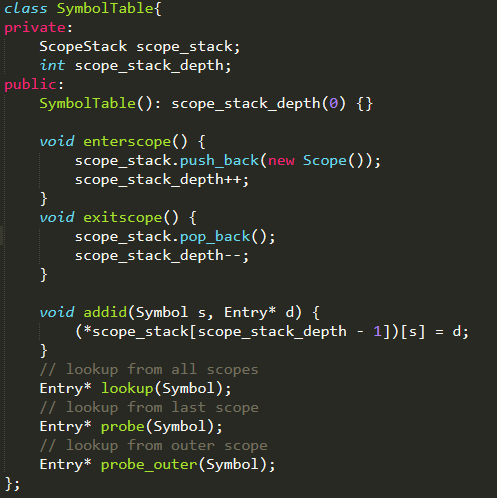
**五、符号表**

symtab.h中提供的符号表是先定义了SymtabEntry，然后通过遍历链表实现的。我尝试基于Map实现自己的符号表。

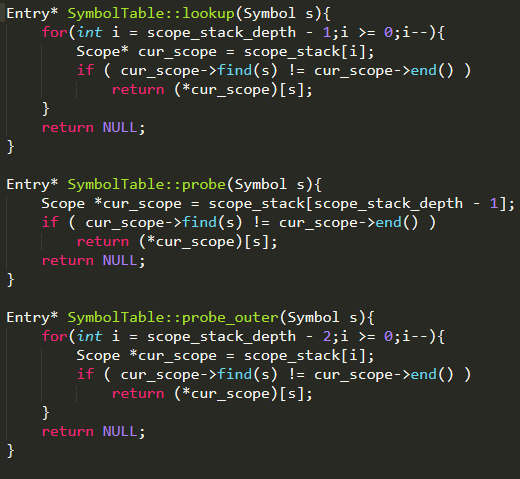
符号表中的一个层级为一个Scope，即如果一个新的类别继承了父类，它就在父类的基础上再添加一个scope。定义一个Scope为一个Symbol : Entry \* 的键值对的字典。Entry的定义可见inclue/PA4/stringtab.h。在类继承的过程中，类层层继承实现了一个栈状结构，称为ScopeStack。



符号表SymbolTable具有一个Scope的栈，同时还有一个整形变量scope\_stack\_depth用以记载当前栈的深度。

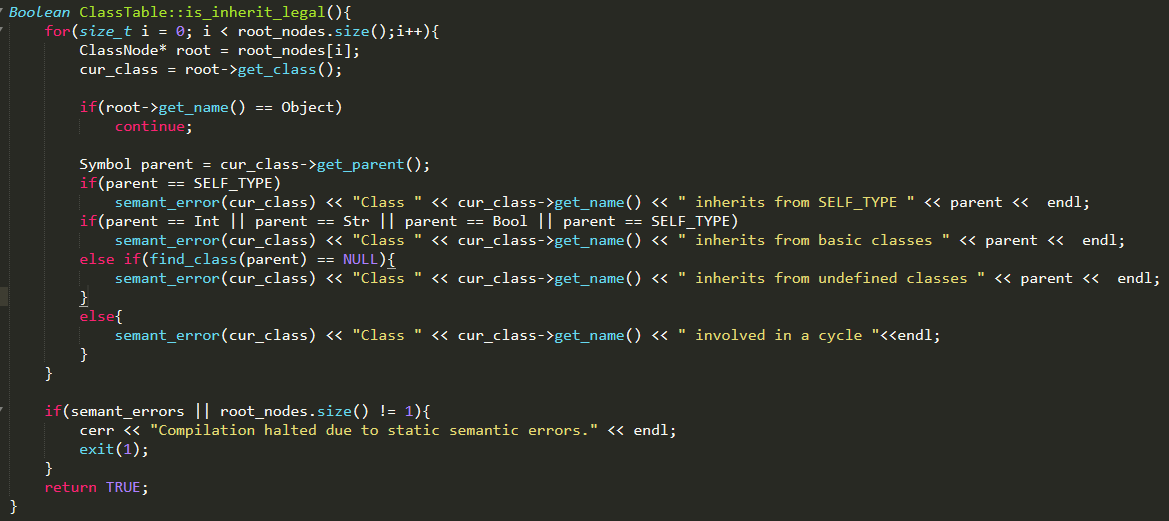


lookup用以在整个符号表进行查找；probe用于在最后一个scope（即子类本身的域）进行查找；在原始的基础上，我又添加了probe\_outer用于在子类所继承的祖先的域进行查找。



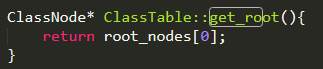
**六、检查继承关系**

在ClassTable中，提供用于检查继承关系的函数is\_inherit\_legal()。在父节点合法且不存在循环继承的情况下，root\_nodes.size()应该为1。如果root\_nodes的数量不为1，就已经足以说明继承不合理了。但为了查明具体是哪里不合理，遍历每个root，如果root是Object，是正常情况。如果root的parent是SELT\_TYPE、基本类、没有定义的类，则是非法情况。而如果root的parent是已经定义的类，那就说明root继承了parent，同时root也是parent的祖先（在类表插入时说明过），即出现了环。

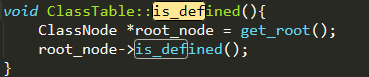


**七、检查是否定义**

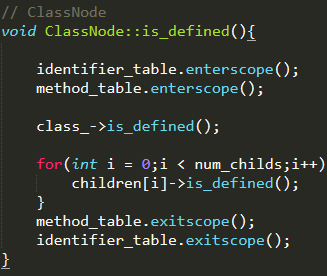
在确认继承关系合理（即只有一个根后），可以使用get\_root()方法来访问继承树的根节点。



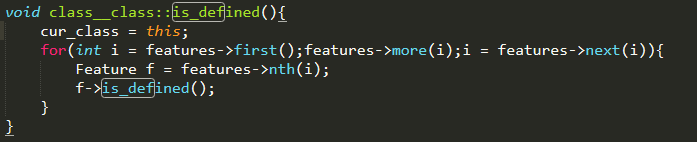
使用is\_defined()函数来检查是否定义。该函数是递归进行的。从ClassTable开始调用该函数。



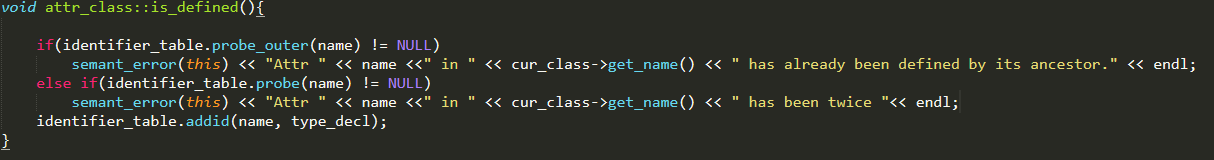
ClassTable的is\_defined调用ClassNode的is\_defined。这里我们使用了2个全局变量的符号表identifier\_table和method\_table，分别用来存放变量标识符和方法的符号表。然后，我们先对类节点的class\_确认是否被定义，再递归调用到子节点上。



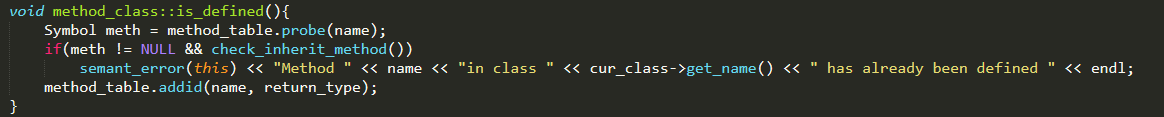
class\_的is\_defined将依次调用其各个feature的is\_defined。

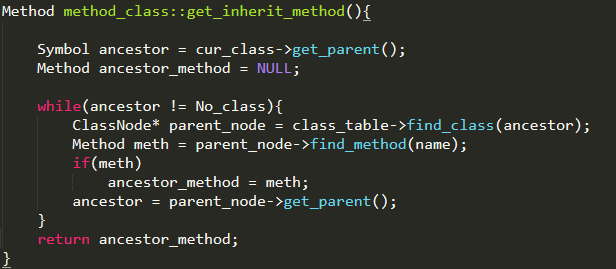


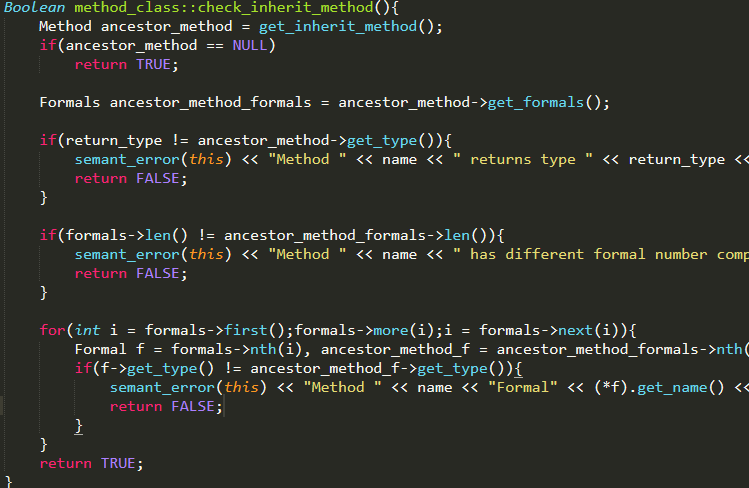
对于attr来说，只要它在identifier\_table中出现过，那么就是重复定义。不过，我们还是要区分，它是在类内重复定义，还是在继承中重复定义。



当对method检查是否定义时，如果当前类定义了该method，则需要检查在method的符号表中是否存在和父类同名的函数，如果存在与父类重名的方法，将依次确认返回类型是否一致，参数个数是否一致，参数类型是否分别一致。

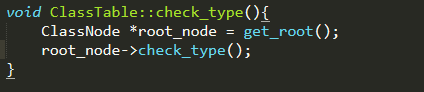




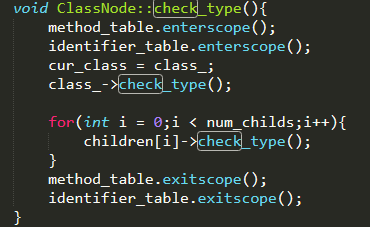


**八、类型检查**

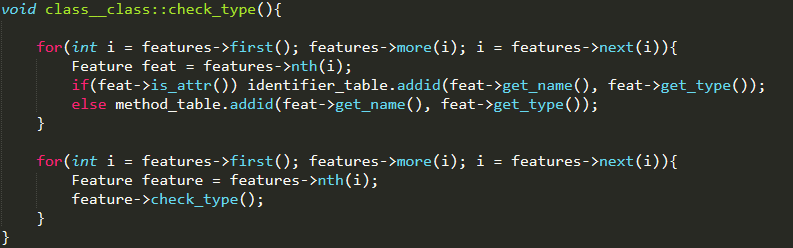
使用check\_type()函数来进行类型检查。和is\_defined一样，该函数也是从类表开始调用，递归到类节点、类、Feature、Formal和表达式上的。



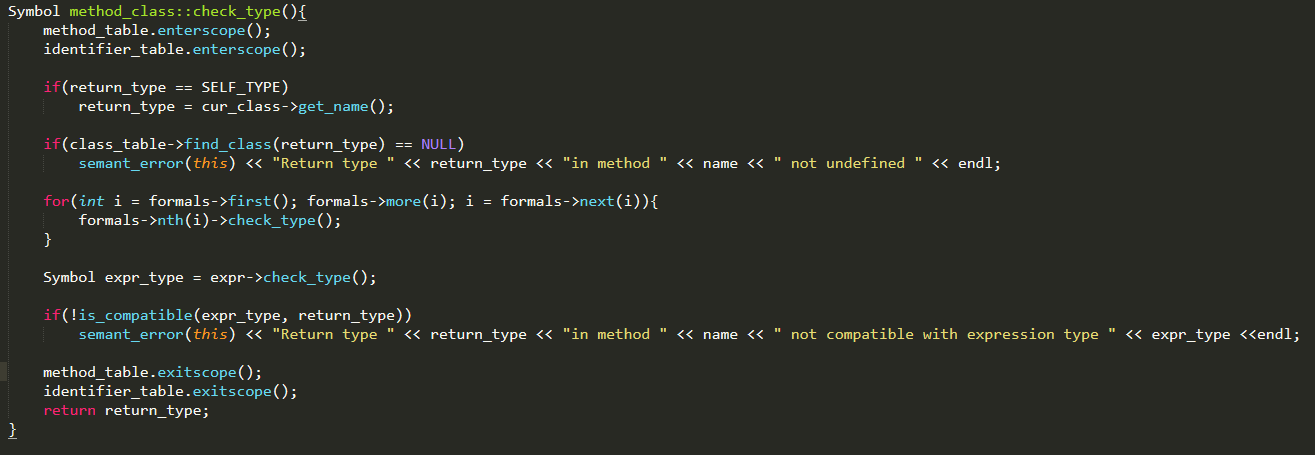
类节点上的check\_type和is\_defined完全一致：调用这个节点对应类的check\_type，然后递归调用子节点的check\_type。



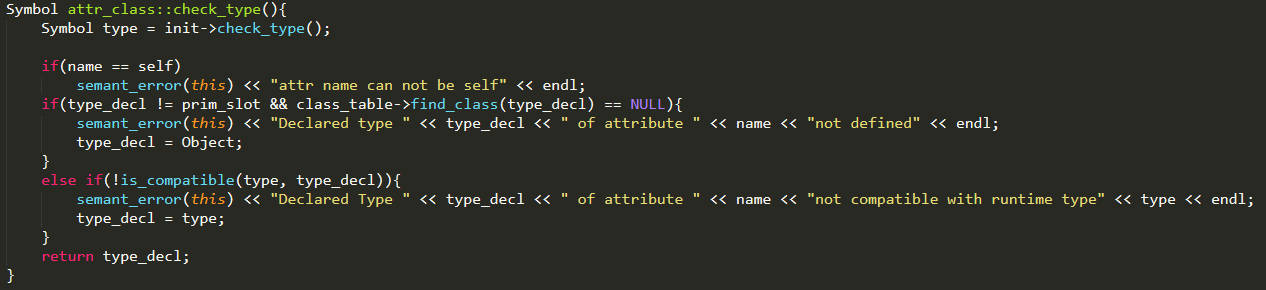
类是由feature构成的，故类的check\_type即依次调用feature的check\_type。在此之前，还需要将feature加入到对应的符号表中。



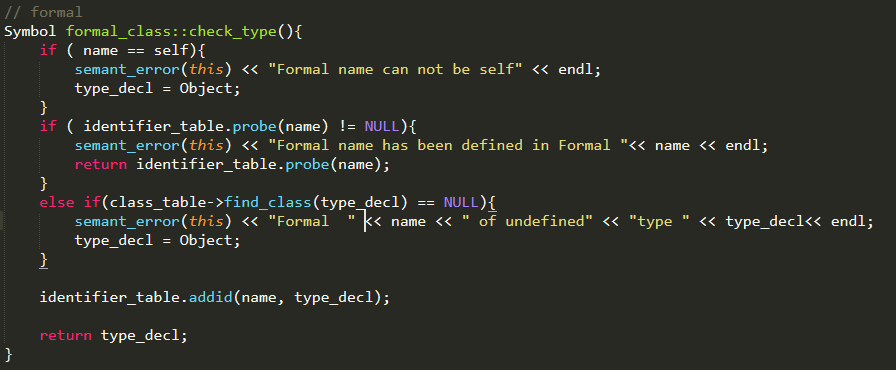
方法的check\_define函数需要先获得return\_type，检查return\_type是否在类表中定义，然后逐个调用formal的检查，再检查表达式返回的类型，判断两个类型是否兼容。判断是否兼容的函数在下面会介绍。



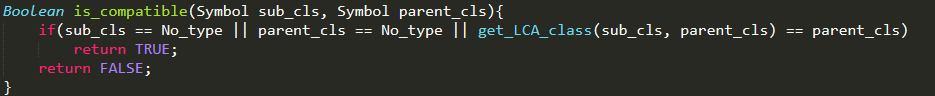
属性的check\_define需要先检查初始化表达式的类型，然后判断该类型是否为self，是否定义在类表中，是否与声明的类型兼容。



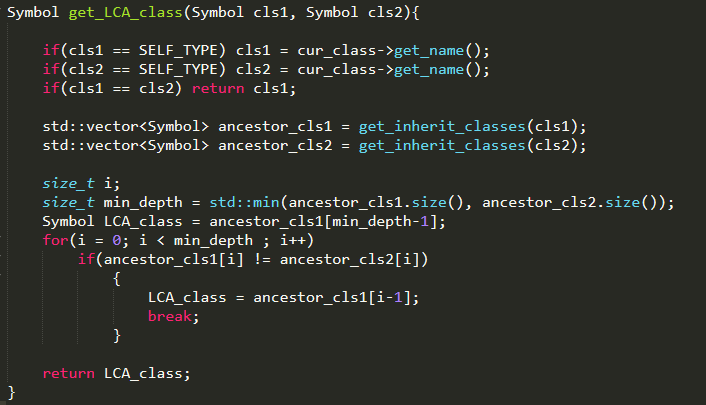
Formal的检查主要是检查该Formal里的标识符名字是否为self、是否重复定义，以及类型是否已经定义。



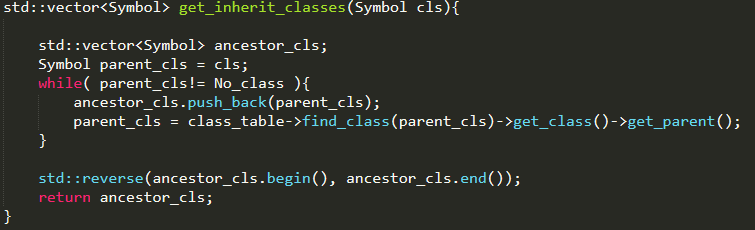
表达式的类型检查中，因为不仅需要判断类别是否相同，还要判断类别是否存在兼容的继承关系。定义接口为is\_compatible。如果两个参数有一个是No\_type，或者两者的LCA（最小公共祖先）即为第二个参数，那么两个参数的类型就是兼容的。



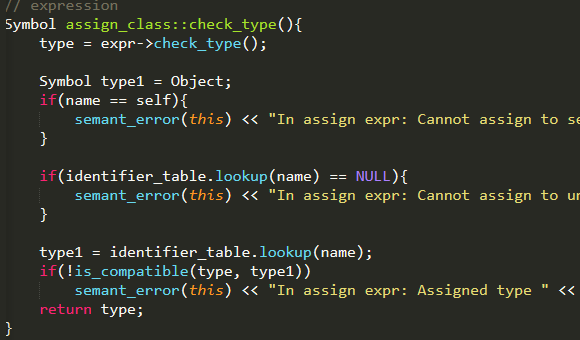
获取LCA的函数如图所示。首先，对两个类别cls1，cls2，分别获取其祖先类别的向量，然后从祖先的根部开始判断两者是否相等。如果两者不存在设定的公共祖先，那么LCA就是Object。如果两者在第i层发现祖先不等，那么在i-1层的祖先就是LCA。如果两者直到最后一层祖先都相等，那就说明两者是同一类别，LCA\_class即类别本身。



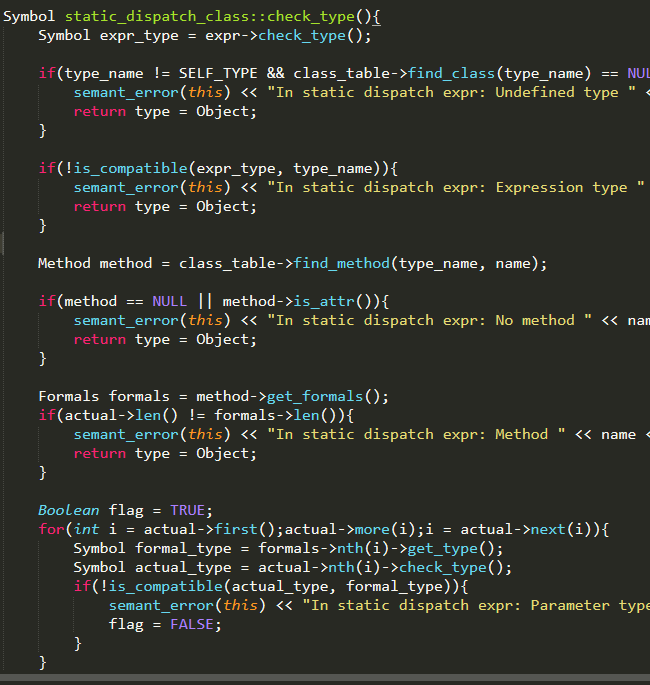
获取祖先类别的函数：



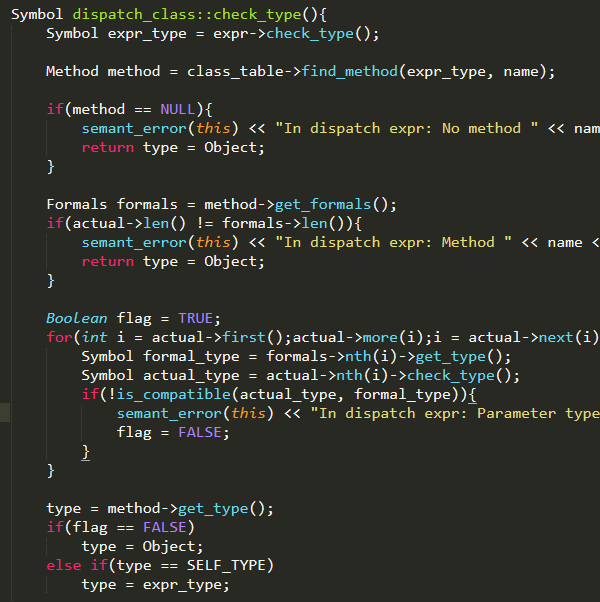
赋值语句首先要检查变量名是否为self，变量名是否定义，赋值变量和声明的类型是否兼容。



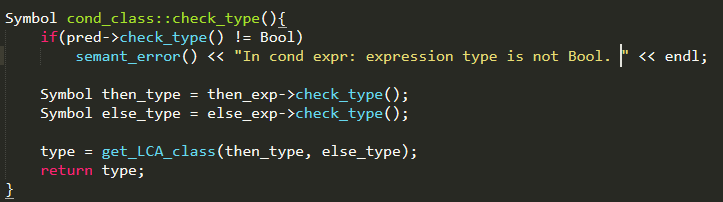
静态的dispatch语句要依次检查：表达式的类型，声明的类是否定义，表达式的类型与声明类型是否兼容，声明方法的是否存在，方法的参数的数量和类型是否一致。返回的类型是方法对应的类型。



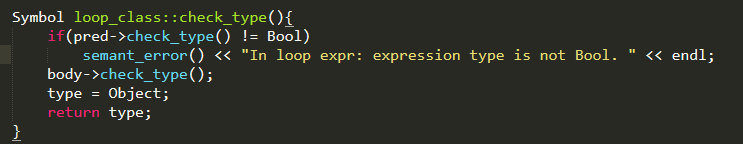
普通的dispatch则不需要检查声明类型是否定义、声明类型与表达式类型是否兼容。使用表达式类型而不是声明类型去检索method。



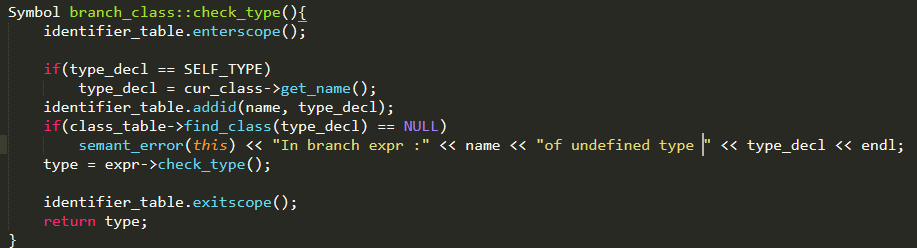
条件语句需要检查表达式类型是否为Bool，然后依次对then表达式和else表达式进行检查。



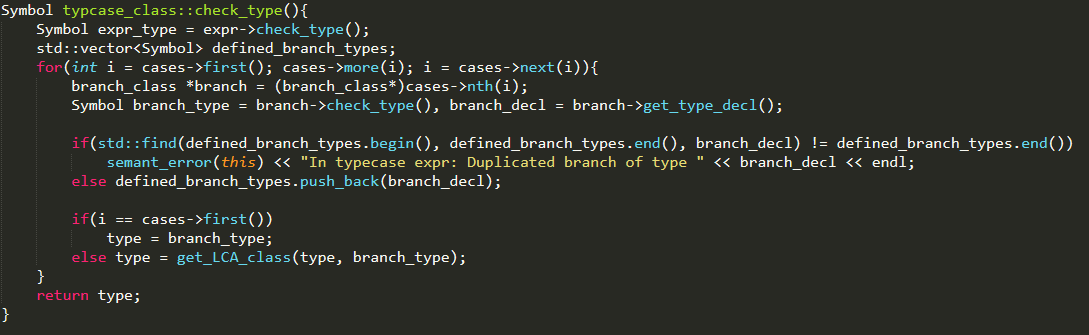
循环语句与其类似。在判断完表达式是否是Bool类型后，对循环体body进行类型检查。



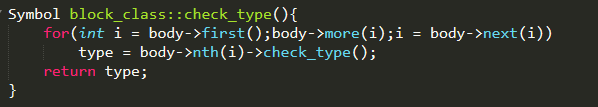
对于typcase语句，需要首先检查每一个branch的语句。每一个branch也是一个scope，需要创建新的标识符符号表。要对branch中声明的类型进行检查是否定义过。之后要返回表达式对应的类型。



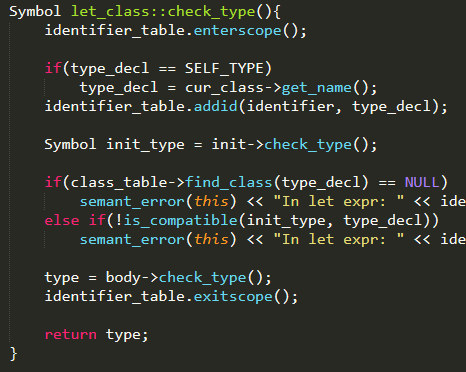
在typcase语句中，一个类型只能对应一个branch，因此我们使用一个向量defined\_branch\_types来记录出现过的branch的类别。如果遇到重复定义，则报错。typcase语句的类型取所有branch的类型的最小公共祖先。



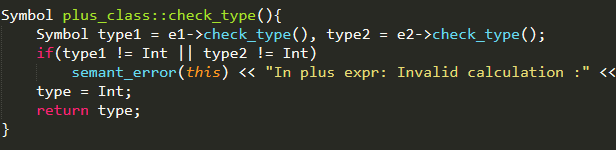
block依次调用body中各个表达式的check\_type即可。



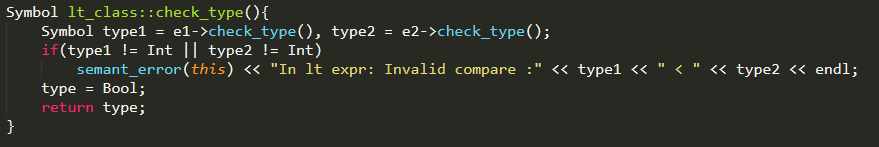
let表达式也需要创建一个新的scope，将声明的类型加入其标识符符号表中，然后确认声明类型是否已经被定义了，表达式的类型与声明类型是否相同。随后，将let语句的body的类型进行类型检查并将结果赋为let语句的类型。



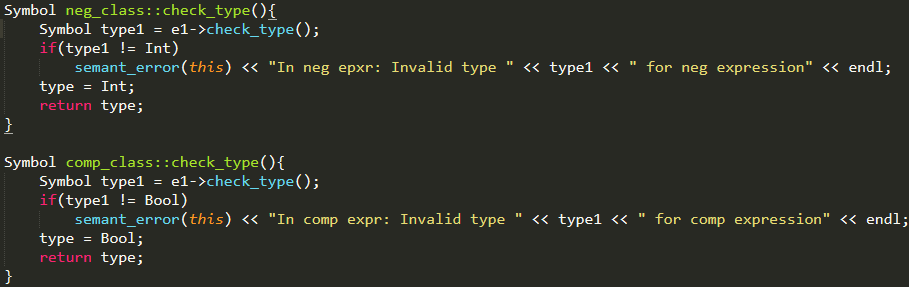
加减乘除四则运算的类型检查均是对2个表达式的类型进行检查是否为int，然后将int作为表达式的类型。



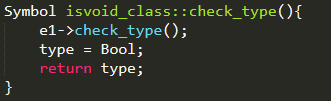
三种逻辑运算lt、leq和eq也是一样的，只是返回值变为Bool型。



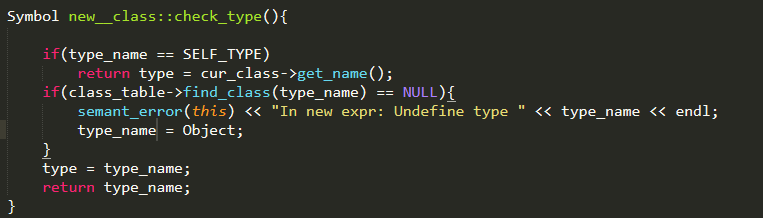
单目运算表达式neg和comp都只需检查参数表达式的类型是否为Int/Bool，并返回相应的类型。



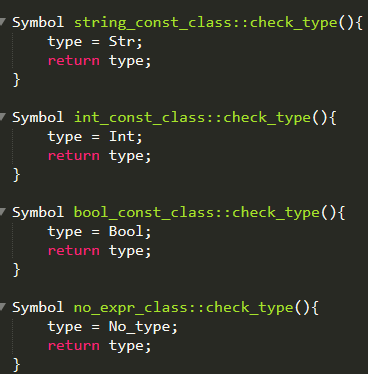
isvoid表达式返回Bool类型，并对表达式进行类型检查。



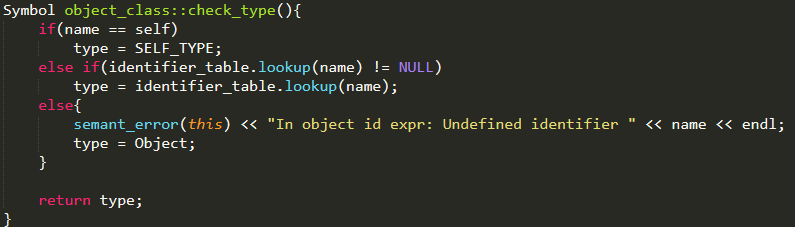
new表达式需要检查类别是否定义了。



string常量、int常量和bool常量的表达式的check\_type函数将type设置为对应的类型即可。no\_expr\_class将type设置为No\_type。

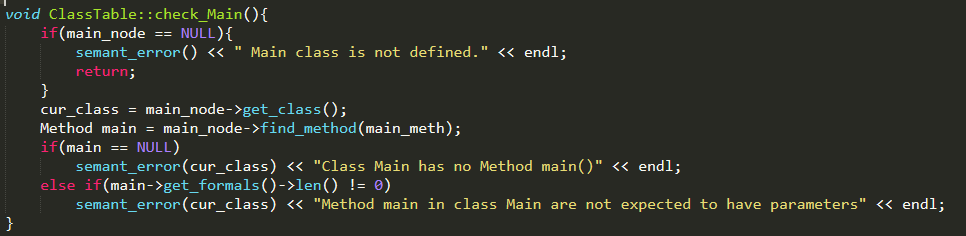


最后是对标识符名字的check\_type。当标识符为self时，返回SELT\_TYPE。否则，在标识符的符号表中查找name，如果没找到则报错，如果找到则返回相应类型。



**九、Main类检查**

获取CaseTable中的main\_node节点。如果该节点为NULL，即不存在Main类。如果存在该节点，则依次检查其是否有函数main()，函数main()是否具有参数。



**十、感想**

PA3无论是从深度和广度来说，都是令人头皮发麻的。与PA1和PA2的工作量相比完全不在同一量级。好在此时对cool已经有了比较深的理解。

从semant()这个起点出发，逐步定义数据结构，实现方法。第一个问题就是数据结构的构建。一开始我希望直接魔改class\_\_class，来构建一个class\_\_class的树，但发现这样写起来比较乱，还要不断更改CLASS\_\_EXTRAS，便定义了ClassNode用以包装class\_\_class，整体的代码逻辑便清晰简单许多。

PA3从难度来说，最难的地方当属类循环继承的检测。一开始，我想尝试用类节点之间进行某种遍历，来检索环的存在。但后来发现，针对于从不在环上的类节点开始的循环继承检查，该方法会陷入死循环。这个难题可以通过使用将类之间的继承关系用多颗继承树组成的森林表示来解决。当存在循环继承时，根据类表上的插入算法，可以保证继承树的根节点一定在这个环上，而最终一个不是Object类的根节点，除非该节点的父类没有定义，否则一定是在某个循环继承当中。

从广度来说，PA3的类型检查涉及到了太多细节，这些需要不断比对阅读cool的语法中仔细的要求，或者在不断debug的过程中才能发现。在debug过程中，我对cool语法的这些细枝末节的点就有了较为深刻的记忆。