实验报告

PA3

1. 实验目标

这一阶段我们需要完成**静态语义检查(Static Semantics)**。上一阶段我们得到了一棵AST，每个结点包含的信息有：

- 这个结点的类型(**这里指Phylum，和下文的类型(Type)不一样**，如Class/Feature/Expression/Plus Expression...)
- 这个结点的位置(在哪一行？)
- 这个结点的构成(比如一个加法表达式，含有两个子表达式)

PA4 Handout中总结了这个阶段需要完成的三个任务：

1.1 检查继承关系

Cool中，`Class`可以通过`Inherits`关键字继承另一个类的内容。我们需要做的就是检查**继承是否是合理的**。简而言之，我们需要**构建一个继承图，让子类指向父类。继承图应该是一个有向无环图(DAG)**，如果我们检测到继承图中出现环，则在语义上，这样的继承关系是不合法的。此外，还需要注意**继承的类是否存在、对基本类的继承情况**等问题。

1.2 检查变量的作用域

每个变量/标识符都有其对应的作用域。我们需要**检查使用的变量名在当前位置是否已经定义了、如果定义，是否出现重复定义**。为此，我们需要在**检查过程中模拟作用域构建与退出的过程**。

1.3 检查表达式类型

在这个过程中，我们需要**遍历整棵AST**，将每个AST结点的类型记录到符号表(Symbol Table)中。然后**根据符号表中已有信息和规定的类型环境(Type Environment)，检查每个结点的类型是否合法**。最后，**为每个AST结点标注(Annotate)类型**。

1.4 结构说明

下面，我们会从**实际实现的逻辑**上出发，依次介绍对Class的检查、对Feature(即Method和Attribute)的检查、对Expression的检查；并在**涉及实验目标的相关部分**指出我们是怎样实现这一部分的检查的。我将代码和测试结果图片都放在了最后面，去除掉这部分内容后，**实际报告篇幅在30页左右**。

2. 对Class的检查

我们语义检查的起点是`program_class`的`semant()`函数。`program_class`是我们在上一个阶段语法分析中的开始符号(Start Symbol)，代表整个程序。它包含了程序文件中的Class列表`Classes`。

对于Class的语义检查项总共如下：

- 检查**类名是否合法**。一些类名不能被使用，如`Int`、`String`、`Bool`、`SELF_TYPE`等。
- 检查**类名是否重复**。Cool语言中，Class不能重复定义。
- 检查**Main Class是否存在，main method是否符合要求**。每个`.cl`文件的执行起点位于**Class Main中的main方法**。必须存在Main Class和其中的main方法，且根据cool manual 9. Main Class中的要求，main method不应该含有参数。
- 检查**继承的类是否合理**。一方面，继承的类应该是已经定义的类；另一方面，继承的类不能是某些规定不能被继承的类，包括：`Int`、`String`、`Bool`、`SELF_TYPE`，见cool manual 8. Basic Classes。
- 检查**继承是否成环**。

PA4 Handout 4. Inheritance中指出，我们应该**先构建一个表示类图的框架，然后分为两部分检查：**

- 先检查**继承相关的合理性**，确保类图（**注意，还包括里面的方法和属性的继承**）是合理的。这样，我们对其他语义的检查就不用考虑相关的合理性，只需要关注语义本身。
- 再检查**其他语义**，如类型检查、属性类型是否定义等。

因此，我们首先实现类图的数据结构，在本次assignment中已经提供了`ClassTable`。我们需要对其进行完善；其次，先进行继承相关的检查，最后检查其他语义。

下面分点叙述相关内容。

2.1 构建ClassTable

我们先思考一下：**我们为什么需要这样一个ClassTable**？或者我们可以思考更本质的一个问题：**我们怎样遍历这些Class(AST结点)**？

我们可以在`cool-tree.h`中看到，每个`Class_`类的成员含有`name`(本类的类名)、`parent`（父类的类名），这是在前一个阶段语法检查中，我们构建AST树就接触到的。如果我们现在有一个Class，我们

就能通过其`parent`，获得其父类的名字(Symbol)。

但是，我们**并没有办法根据Symbol找到Class**。这正是为什么我们需要相应的数据结构。通过我们的`ClassTable`，我们应该**能从一个类访问到其父类，直到顶部；也能根据提供的类名，判断这个类名是否合法**。

根据以上描述，我们的`ClassTable`应该实现为**一个从名字(Symbol)到类(Class)的映射**。在C++中，这可以用**map**来实现，在`semant.h`中的`ClassTable`中定义如下：

```
std::map<Symbol,Class_> classTable;
```

在`semant.cc`的`ClassTable`构造函数中，我们需要将给定的`classes`加入到`classTable`中。这里，`Classes`是一个提供的`ListNode`（链表）类，遍历方法参考cool-tour 6.2，如下：

```
for(int i =classes->first();classes->more(i);i=classes->next(i))
{
    curr_class = classes->nth(i);
    classTable[curr_class->get_name()] = curr_class;
}
```

这里,`get_name()`是我们额外定义的方法，是一个简单的getter，写在`cool.handcode.h`中。之后我们还会在`cool.handcode.h`中定义很多这样的函数，因为数量较多，且功能简单（如下），就不展示详细代码了；`curr_class`是一个`ClassTable`中的全局`Class_`全局变量，主要目的是为了**处理`SELF_TYPE`**，后面会详细说明。

```
#define Class__EXTRAS                                 \
...
virtual Symbol get_name() = 0;


#define class__EXTRAS                                  \
...
Symbol get_name() { return name; }
```

在构建ClassTable的过程中，我们就应该完成部分检查。因为这些**不合规的类不应该被加入到`ClassTable`中**。

具体的检查项分为以下几项：

- 如果当前class是`Int`、`String`、`Bool`、`IO`、`Object`、`SELF_TYPE`，这个Symbol不属于一个具体的类，不能在class定义的时候出现，报错。

- 如果当前class已经在`classTable`中，说明当前正在重定义，报错。

此外，我们还需要将Basic Classes添加到`classTable`中，`install_basic_classes()`已经定义了五个Basic Class，我们可以直接在其中将它们加到`ClassTable`中。见代码8.1。

这里说明一下关于`semant_error()`相关的内容。它是我们自定义的报错函数，接收一个Class，并将**类的当前文件名(应该是用于多文件编译的，我们不涉及）和当前AST的行号（这正是我们在词法分析和语法分析阶段处理的）**输入到错误流。之后，我们还需要定义其他形式的`semant_error()`函数，因为需要报错的AST不一定只有Class，还包括Feature、Expression等，具体见问题6.4。另外，这里还有一个变量`ifStop`。它是一个Bool变量，用于决定**分析过程是否停止**。这属于一些实现上的细节，在这里不详细说明，见相关部分。

2.2 检查继承

接下来我们进行检查的第一步：**检查继承相关内容**。具体如下：

- 检查继承的类是否存在。
- 检查继承的类是否是`Int`、`Bool`、`String`、`SELF_TYPE`。
- 检查继承图中是否有环。

对于检查继承的类，我们可以**遍历构建好的`classTable`，逐个检查每个Class的`parent`是否在`classTable`中(或者是否是指定的类)即可**。

而对于继承图中是否有环，我们考虑**为每个检查的类维护一对快慢指针，每次慢指针往图上前进一次（获取父类），快指针前进两次（父类的父类）。当快指针或慢指针碰到Object时，就直接返回，说明图中没有环；否则，当快慢指针相遇时，说明图中有环**。

我们实现`void checkInheritance()`函数于代码8.2

注意点如下：

- 父类合理性检查到Object类时，其父类为`No_type`。我们应该直接跳过。
- 继承图是否有环检查到Object类时，直接跳过。
- 我们这里是**逆序遍历每个类**。具体原因我其实不清楚，因为官方的语义解释器实现的效果是这样的，猜测可能是因为：用户习惯上，一般**按照继承顺序从上到下书写类，所以逆序检查效率更高**。另外这里检查结果上，由于map数据结构自动排序的特性带来了麻烦，具体见问题6.6。

3. 对Feature的检查

在完成了对Class的检查后，接下来我们进行对每个Class内部的Feature，即属性(Attribute)和方法(Method)的检查。

对属性，我们需要检查的内容如下：

- 检查属性**声明的类型是否定义**。
- 检查**当前类内**，是否**重定义属性**。
- 检查**是否重定义继承的属性**。
- 检查属性**声明的类型是否与初始化表达式的类型相合(conform)。**
- 检查属性名是否为**self**。

对方法，我们需要检查的内容如下：

- 检查**当前类内**，是否**重定义方法**。
- 检查**重载是否合理**。包括重载返回类型必须一致、参数类型必须一致等。
- 检查**参数是否合理**。包括参数类型必须存在、参数不能重定义等。
- 检查**返回类型是否合理**。包括返回类型必须存在、返回类型是否与表达式类型相合等。

此外，我们还要关注**作用域的问题**。这是什么意思呢？我们举个例子，如下面这段Cool程序：

```
Class A{
    x:String;
    y:Int;
    func(x:Int):Int{
        x + y
    };
};
```

如果我们不对作用域进行处理，那么，此时就出现了两个问题：

- func()中的x，是Int类型(即参数x)，还是String类型(即属性x)?
- func()中的y，此时是否有定义?

处理作用域就是要让我们在程序的每个地方都能正确获取每个标识符的含义，即是否有定义，如果有定义，用哪个定义。

下面，我们先介绍怎么处理作用域，然后分别说明如何检查属性和方法，并在检查的过程中处理作用域。

3.1 作用域处理：Symbol Table

cool-tour 4.中为我们提供了Symbol Table，这是用于实现**作用域的数据结构**，其内容是**作用域的链表(lists of scopes)**，每个**作用域又是<标识符,数据>对的链表**。具体而言，在Symbol Table这个链表中**靠前的作用域是更内部的**，而**靠后的作用域是更外部的**。具体实现位于`symtab.h`中。

下面介绍几个常用的操作：

- `enterscope()`:创建一个新的scope(链表)到**头部**(因为是更"新"的定义。`Symbol Table`是使用头插法构建的链表)。
- `exitscope()`:弹出头部的scope，即**当前scope**。
- `addid(s,i)`:将Symbol `s`和数据`i`添加到**当前scope**。
- `lookup(s)`:从头部开始，查找`s`对应的第一个scope内的信息。(也即`s`对应的最内部scope)

我们可以看到，Symbol Table很好地模拟了作用域构建与退出的过程。例如，上面举的例子程序可以以如下的形式**利用Symbol Table模拟作用域构建、使用与退出的过程**。

```
Class A{
    // 进入Class A,把它的属性加入当前scope
    // SymbolTable: {x:String, y:Int}
    x:String;
    y:Int;
    // 进入method func,把它的参数加入当前scope
    // SymbolTable: {x:Int} <- {x:String, y:Int}
    func(x:Int):Int{
        // 此时我们用lookup函数查找x和y的类型
        // 发现x是Int类型，y是Int类型
        x + y
    };
    // 退出method func,弹出它的scope
    // SymbolTable: {x:String, y:Int}
};
// 退出Class A,弹出它的scope
// SymbolTable: 空
```

需要注意的一点是，与`ClassTable`不同，`SymbolTable`只是一个**辅助性质**的数据结构，它**保存的都是中间数据**，记录了在分析程序过程中作用域的变化，**并不需要作为结果来保存，结束时它应该是空的。**

PA4 Handout 5. Naming and Scoping中提到了Cool中四种**引入新标识符的方法**，分别为：

- Class中的**属性**定义。
- 方法的**形式参数**(formal parameter)。

- let表达式。
- case语句的分支。

我们需要在分析到这四种情况时，对作用域进行修改，包括什么时候构建新作用域、什么时候退出当前作用域。此外，还要特别处理的一个情况是**继承类的属性**，即**在子类的作用域中，其祖先类的属性也是可用的**。

在我们的PA中，`SymbolTable`应该是怎样的一个形式呢？就如上面所说的，我们需要知道每个标识符的**类型**，这是因为我们进行的是**静态语义检查**，暂时只需要关注类型(参考cool-manual 12.1 Type Environments)。因此，我们的`SymbolTable`应该实现如下：

```
SymbolTable<Symbol, Symbol> objectEnvironment;
```

表示每个scope的每个entry由<标识符名字，标识符类型>构成。

3.2 构建Method Table和Attribute Table

我们可以通过自定义的getter函数，从Class中获取该Class的Features，然后判断是否是Method（**通过在Feature_Class的两个子类Method_Class和Attr_Class中重载同名函数**），并检查对应参数，如下：

```
#define Feature_EXTRAS                                  \
...
virtual bool is_method() = 0;                           \
virtual bool is_attr() = 0;

#define method_EXTRAS                                  \
...
bool is_method() { return true; }                      \
bool is_attr() {return false; }

#define attr_EXTRAS                                   \
...
bool is_method() { return false; }                    \
bool is_attr() { return true; }
```

如何检查每个Class里的Feature，在前面的`check_main()`函数中已经展示了，即：通过Class获取其Features，然后遍历，检查每个Feature，并判断其是Method还是Attribute，再分别检查。这里，为了实现上方便，我仿照`classTable`构建了`methodTable`和`attrTable`，如下：

```
std::map<Class_,std::vector<method_class* >> methodTable;
std::map<Class_,std::vector<attr_class* >> attrTable;
```

每个Table存储了`Class_`到其对应的methods/attrs的映射，这样在后面的检查和构建中，会方便很多。

从直观上想，构建这样的Table其实是很简单的：我们只需要遍历每个Class，获取其Feature，分类处理加到对应的Table即可。但我这部分实际上的实现是十分复杂的，主要是为了与官方语义解释器的表现保持一致，详见（问题与解决）

简单解释一下官方语义解释器的逻辑：在检查每个类时，我们**获取其继承路径，从Object开始，检查路径上的每个Class**。在检查每个Class时，**先检查这个Class的所有祖先的方法和属性，如果与当前类的方法和属性重名，方法→检查重载；属性：报错；再检查当前类内是否有方法和属性重名，方法和属性→报错**。通过这样的检查，一方面，我们**构建了每个类的合法方法和属性**；另一方面，我们保证了**类继承时方法和属性的正确性**，这和PA4 Handout 4. Inheritance中要求的**先检查继承相关**的要求是一致的。这一段逻辑十分重要，在后面的大量测试中都证明了该检测逻辑的正确性，见5.测试样例。问题6.5中，我对这段检查逻辑和这样检查的好处做了详细的分析。

首先说明**如何获取继承路径**，即**如何检查一个类的祖先类**。类似检查继承性函数`check_inheritance()`的实现，只需要不断获取一个类的父类直到Object即可。具体实现见代码8.3

代码见代码8.4。这个实现很复杂，但思路上并不难，主要利用了一个从`Class_`到`bool`的map记录某个类是否被已经安装/检查过，并在继承路径上进行进一步检查。另外，这里还有一个`repeat`变量，这是因为**属性/方法的重复定义有检查的优先级，优先检查继承类再检查自身（或者说，是“自顶向下”检查的）**。注意我们**只将那些合法的属性和方法记录下来**，因为经过我的多次测试与对比，**检查到错误的方法/属性定义相当于被忽略了，在后面的检查中不会涉及到它们**。例如：

```
Class A{
    a:INT;
}

Class B inherits A{
    a:Int;
};
```

这里B中就不会报错，因为A中的定义是错误的，相当于不存在。

这里还涉及到一个如何将`Feature`转换为`method_class`或`attr_class`的问题。因为前者是后两者的父类，按理说是不行的。但**使用指针类型就能强制转换**，这也是为什么我们保存的vector是指针数组。

在这段程序的最后我们单独进行了对Main的检查：是否定义Main Class、main method、main method是否有参数。在这里检查，一方面是为了与官方语义解释器的顺序相适应；另一方面，在完成

了classTable、methodTable和attrTable的构建后，检查这些就变得十分简单。

另外还有一点是，属性名**不能为self**，也进行了单独的检查，且不能将其加入到attrTable中。

3.3 对Attribute的检查

上一步我们只是完成了对每个Class内有哪些Method和Attributes的记录，且只检查了重复定义的情况。接下来，我们对属性和方法进行剩下的检查。注意实际上**属性和方法是一起检查的，因为我们是按照Feature出现的顺序来依次检查的**，这样，报错顺序与程序的顺序才能比较好地保持一致。只是我们分开说明这两者的实现逻辑。

详细地，对于属性需要检查的内容有：

- 检查属性**声明的类型是否定义**。
- 检查**当前类内**，是否**重定义属性**。(已经检查)
- 检查**是否重定义继承的属性**。（已经检查）
- 检查属性**声明的类型是否与初始化表达式的类型相合(conform)。**
- 检查属性名是否为**self**。（已经检查）

3.3.1 填充Class作用域

在前面已经实现了对每个Class的属性重复性检查，已经确定了，Class的每个属性在**继承性**上都是合法的。在进行接下来的检查之前，我们需要处理作用域的问题：因为在每个类内，**其属性，及其祖先类的属性都是可用的**。我们需要将这些属性添加到对应的scope中。

填充作用域代码见代码8.5。每次考察一个类时，获取其继承路径，将路径上所有的属性加入到当前scope中(注意我们添加的属性都是已经检查过的合法属性)。

这里有一个实现上的细节：**仅在考察的每一个类创建新scope，而不在考察其继承路径时也创建新scope**。这是因为，`SymbolTable`的`addid()`方法是头插法，更新的内容会自动被加到当前scope的更前面，而我们考察继承路径的方法是**从Object到当前类**。我们举个例子：

```
Class A inherits B{
    a:Int;
    b:Int;
};

Class B{
    a:String;
    b:Bool;
    c:Int;
};
```

则我们在考察A时，构建的SymbolTable为：

$$\{b:\text{Int}, a:\text{Int}, c:\text{Int}, b:\text{Bool}, a:\text{String}\}$$

而实际情况下，SymbolTable的表示形式应该为：

$$\{b:\text{Int}, a:\text{Int}\} \leftarrow \{c:\text{Int}, b:\text{Bool}, a:\text{String}\}$$

对于`lookup()`函数而言，结果没有任何区别。因此这样的实现属于更简单的一种方式，不需要在继承路径内部多次创建scope和退出scope(且我个人认为这也是对**单个类作用域**的更好描述)。

3.3.2 对声明类型存在性的检查

声明类型存在性的检查十分容易，在`classTable`中查找即可。代码见代码8.6。
这里有几点需要注意：

- `checkType()`是用于**检查表达式类型**以及**标注表达式类型**的函数。我们在4.对Expression的检查中详细说明和实现。**注意我们是在正确的scope中进行检查的，这很重要，后面也有不少类似的情况。**
- `checkType()`函数应该**尽量晚地调用**，即**只有在需要用的时候才用，不要提前检查**。这样做是为了与官方语义解释器的报错顺序保持一致。
- 如果**声明类型不存在**，就不要进行后面的相合性检查了。
- 这里含有对SELF_TYPE的检查。关于SELF_TYPE的介绍详见Cool manual 4.1 SELF_TYPE。简而言之，可以简单理解为一个指代当前类的名字（实际上不是）。在这里，我们将SELF_TYPE转换为实际的类名以便进一步处理。
- `prim_slot`是在`install_basic_classes()`中，`Int`类和`Bool`类含有的属性类型，它不需要被检查。

3.3.3 对相合性的检查

对属性的检查还剩下最后一项：**声明的类型是否与初始化表达式的类型相合(conform)**。关于相合的介绍可见Cool manual 4 Types。简而言之，如果类型A能够“赋值”(包括赋值操作、函数返回、属性初始化等)给类型B，那么**类型A必须是类型B的子类(或本身)**，这是，因为子类相比父类有更丰富的信息，这样的赋值是更合理的。

为了实现这样的检查，我们就需要在给定两个Class的情况下，检查一个是否是另一个的祖先。通过`getInheritancePath()`函数就很容易实现：在一个类的继承路径上查找另一个类即可。实现见代码8.7。

这里需要注意的是**对SELF_TYPE的处理**，因为SELF_TYPE可能作为函数返回类型、属性声明类型等出现。Cool manual 4.1 SELF_TYPE中提到有：

$$C \leq P \rightarrow \text{SELF\_TYPE}_C \leq P$$

简而言之，**在检查相合性时，我们可以把所有SELF_TYPE换成当前类来检查（前提是SELF_TYPE是出现在不等号左边的）**。

最终对属性的相合性检查见代码8.7

需要注意没有初始化的情况，此时不进行检查。

3.4 对Method的检查

对Method的检查具体如下：

- 检查**单个类内，是否重定义**。（已经检查）
- **重载**时，检查**返回类型是否一致**、**参数个数是否相同**、**参数类型是否相同**。
- 检查参数**类型是否为SELF_TYPE**。
- 检查参数**类型是否已定义**。
- 检查**参数名是否为self**。
- 检查参数**是否重定义**。
- 检查**返回类型**是否定义。
- 检查**返回类型**是否**与表达式类型相合**。

此外，我们还要为Method的**添加作用域**：进入Method时，创建作用域，并将**方法参数**加入到作用域；退出Method时，同样退出作用域。

3.4.1 检查重载

当一个Class内对一个**继承类的方法**重定义时，称为重载。我们在构建Method Table时就已经检查了重载，只不过前面只留下了函数接口，没有说明重载检查的实现形式。重载的要求在上面已给出，下面我们依次检查：

- **返回类型是否一致**。
- **参数个数是否相同**。
- **参数类型是否相同**。

需要注意的是上述检查的优先级从上到下，这是检查难易程度决定的（与官方实现保持一致）；另外，参数类型是否相同的要求也**隐含了顺序必须是相同的**。

给定一个Method，我们可以通过自定义的getter获取其成员`return_type`、`expr`、`formals`和`name`。发生重载时，我们直接**检查两个方法的返回类型、参数个数，并依次遍历两者参数，检查是否完全一致即可**。代码实现见代码8.8。

3.4.2 检查参数

接下来检查的是Method的`formals`成员。还是对`formals`进行遍历，考察每一个成员。检查内容如下：

- 检查参数**类型是否为SELF_TYPE**。
- 检查参数**类型是否已定义**。
- 检查**参数名是否为self**。
- 检查参数**是否重定义**。

这些检查的实现都很直白，不再叙述其实现逻辑。具体实现见代码8.9。

有一点需要注意：**我们需要把参数加到当前方法的作用域中，且一定注意：是在检查方法的表达式类型之前**。因为表达式中**可能使用了方法的参数**，这个顺序**不能颠倒**。

3.4.3 检查返回类型

最后，我们检查方法的返回类型，包括：

- 检查**返回类型**是否定义。
- 检查**返回类型**是否**与表达式类型相合**。

返回类型定义的检查十分简单；而检查相合性，首先需要获得表达式的类型，这是通过：

```
method->get_expr()->checkType()
```

来完成的。`checkType()`我们上面也碰到过一次，用于**检查、标注表达式的类型**，在第4部分中进行详细实现。再次强调：`checkType()`需要在**正确的作用域下使用**。

这两项检查的实现见代码8.10，其中还包含了对其他函数的调用和作用域模拟的实现。在整体结构上，有几点需要注意：

- `enterscope()`和`exitscope()`是成对出现的，所以中间**不能有直接return之类的情况**，否则会导致创建的作用域没有正确退出。
- 按照官方语义解释器的表现，应该**先检查返回类型，再检查表达式类型**。
- 返回类型没定义时也要检查表达式，在检查相合性时再直接跳过。
- 这里有一个特殊情况是没有表达式时，其类型是`No_type`，也直接跳过相合性检查。另外，后面的4.15 let表达式与这里也有关。

4. 对Expression的检查

我们已经完成了对Class、Method和Attr的检查。接下来，我们需要对**更小的单位：表达式**进行语义检查，检查**类型是否合理**，并**为其标注合适的类型（包括错误处理）**。

作为AST结点的表达式是由小到大复合构成的。前面我们已经涉及到：**属性的初始化表达式、方法内部表达式**的类型检查。这些表达式又可以由更小的表达式构成，例如：

```
Class A{
    a:Int <- 1 + 1;
    func(x:Int,y:Int):Int{
        {
            x <- 2 * 3;
            y <- 8 / 4;
            x + y;
        }
    }
};
```

这个例子中，初始化属性a的表达式是一个**加法表达式**；而方法func()的表达式是一个**block表达式**，block表达式又由**两个赋值表达式（右侧是乘法/除法表达式）和一个加法表达式构成。**

对Expression的检查和标注，就是**对每一种类型的表达式根据已知上下文信息（该表达式的成员，当前作用域的可用标识符信息）和类型规则（Type Rules）来决定其类型，并在遇到错误时进行合适的报错和错误处理。**最终，**从小到大，构建整个程序的全部类型检查和标注。**

下面，我们先介绍**类型规则**和**错误处理**，再**由易到难**，说明每个表达式的类型检查方法。

4.1 类型规则(Type Rules)

简单来说，类型规则就是**在满足给定条件下，该表达式的类型应该是什么。**例如：

$$\frac{\begin{array}{l} O, M, C \vdash e_1 : Int \\ O, M, C \vdash e_2 : Int \\ op \in \{*, +, -, /\} \end{array}}{O, M, C \vdash e_1 \ op \ e_2 : Int}$$

横线上面的称为**假设(Hypothesis)**，表示：**如果满足上面这些假设，那么横线下面的式子就为真（注意，反过来不成立）**。每个式子中，O代表**对象环境(Object Environment)**，也就是我们构建的符号

表，保存了**当前作用域下的可用标识符及其类型**；M代表**方法环境(Method Environment)**，表示**当前可用的方法及其参数信息、返回类型等**（我们没有专门实现这样的环境，但在需要时，可以**通过获取继承路径检查methodTable**简单地获得需要的方法）。C代表**当前类**，也就是我们保存的curr_class，主要用于处理SELF_TYPE。

这项规则的意思就是：如果两个操作数都为Int类型，且操作符为算数运算符，则对应的算术表达式的类型是Int。

4.2 错误处理

当类型规则的假设不满足时，我们应该为该表达式标注什么类型？PA4 Handout 6.Type Checking中的建议是这样的：

> A simple recovery mechanism is to assign the type Object to any expression that cannot **otherwise** be given a type.

乍一看我们应该为所有出错的表达式都标注为类型Object。但这句话的重点在于otherwise这个词：实际上，这句话的意思是，**只有在没有类型可标注，或者按照规则要标注的这个类型没有定义时，才标注Object；否则，按照原来的规则标注类型**。这样的错误处理表现与官方的语义解释器表现是一致的，见问题6.1。

4.3 常量表达式和无表达式

$$\frac{}{O, M, C \vdash true : Bool} \quad [\text{True}]$$

$$\frac{}{O, M, C \vdash false : Bool} \quad [\text{False}]$$

$$\frac{i \text{ is an integer constant}}{O, M, C \vdash i : Int} \quad [\text{Int}]$$

$$\frac{s \text{ is a string constant}}{O, M, C \vdash s : String} \quad [\text{String}]$$

常量表达式不需要任何假设，可以直接返回。它们属于最基础的表达式；至于无表达式，我们在语义分析阶段构建AST时使用过，它应该返回`No_type`，并在类型比较时进行特殊处理。

代码实现见代码8.11。注意`checkType()`函数不仅要返回需要的类型（**这是为了利用子表达式的类型**

构建更大表达式的类型检查），还需要对类型(Expression类型的type变量)进行标注，这会在结果的AST树上显示出来。

4.4 算数表达式和比较表达式

$$\frac{O, M, C \vdash e_1 : Int \quad O, M, C \vdash e_2 : Int \quad op \in \{*, +, -, /\}}{O, M, C \vdash e_1\ op\ e_2 : Int} \quad [\text{Arith}]$$

$$\frac{O, M, C \vdash e_1 : Int \quad O, M, C \vdash e_2 : Int \quad op \in \{<, \leq\}}{O, M, C \vdash e_1\ op\ e_2 : Bool} \quad [\text{Compare}]$$

我们在上面已经介绍过算数表达式的规则了，实现也非常简单：调用两个子表达式的`checkType()`函数，进行对比即可；比较表达式的规则也基本完全一致。

代码见代码8.12(因为形式都一样，只给出加法表达式的实现为例)。当出现错误时，需要进行对应的报错（不需要修改类型，表达式仍为Int或Bool类型）。报错情况和注意点如下：

- 操作数不都为Int时，报错。

4.5 逻辑取反和算数取反表达式

$$\frac{O, M, C \vdash e_1 : Bool}{O, M, C \vdash \neg e_1 : Bool} \quad [\text{Not}]$$

$$\frac{O, M, C \vdash e_1 : Int}{O, M, C \vdash \tilde{}\, e_1 : Int} \quad [\text{Neg}]$$

检查子表达式类型是否为Bool或Int即可。实现见代码8.13。报错情况和注意点如下：

- 操作数不为Bool(not表达式)/Int(neg表达式式)时，报错。

4.6 isvoid表达式

$$\frac{O, M, C \vdash e_1 : T_1}{O, M, C \vdash \text{isvoid } e_1 : Bool} \quad [\text{Isvoid}]$$

isvoid表达式总是返回Bool类型。需要注意的一点是，**尽管用不到e_1表达式的类型，我们仍然需要调用e_1表达式的`checkType()`函数**。这是因为我们需要对程序的所有表达式都进行类型的检查和标注，如果e_1表达式内部出错，就要报对应表达式的错误。

代码实现见代码8.14。

4.7 等于表达式

$$\frac{\begin{array}{l} O, M, C \vdash e_1 : T_1 \\ O, M, C \vdash e_2 : T_2 \\ T_1 \in \{Int, String, Bool\} \vee T_2 \in \{Int, String, Bool\} \Rightarrow T_1 = T_2 \end{array}}{O, M, C \vdash e_1 = e_2 : Bool} \quad [\text{Equal}]$$

等于表达式的要求比较特殊：

- 当两个表达式的类型都不是Int/String/Bool时：直接标注类型Bool并返回；
- 当至少一个表达式类型为Int/String/Bool时：必须**两个表达式类型相同**，否则需要报错。

代码实现见代码8.15。报错情况和注意点如下：

- 类型不满足上述要求时，报错。

4.8 循环表达式

$$\frac{\begin{array}{c} O, M, C \vdash e_1 : Bool \\ O, M, C \vdash e_2 : T_2 \end{array}}{O, M, C \vdash \text{while } e_1 \text{ loop } e_2 \text{ pool} : Object} \quad [\text{Loop}]$$

同样的,loop表达式也需要调用两个子表达式的`checkType()`(尽管e2的类型用不到)；然后返回Object

类型。

代码实现见代码8.16。报错情况和注意点如下：

- e_1类型不为Bool时，报错。

4.9 block表达式

$$\frac{\begin{array}{c} O, M, C \vdash e_1 : T_1 \\ O, M, C \vdash e_2 : T_2 \\ \vdots \\ O, M, C \vdash e_n : T_n \end{array}}{O, M, C \vdash \{\ e_1; e_2; \ldots e_n;\ \} : T_n} \quad [\text{Sequence}]$$

block表达式将最后一个表达式的类型作为自己的类型，只需要遍历block的Expressions，并对每个表达式调用`checkType()`即可。这里要注意：**可能有最后一个表达式的类型是一个未定义类型的情况**。不过在后面我们会对所有**直接**产生未定义类型的表达式进行错误处理，从而这里就不需要做任何特殊处理了。

代码实现见代码8.17。

4.10 object表达式

$$\frac{O(Id) = T}{O, M, C \vdash Id : T} \quad [\text{Var}]$$

object表达式就是**获取标识符类型的方式**，也是我们在前面构建Symbol Table的原因。这也是一个最基本的表达式之一。我们应该在当前的Symbol Table中通过`lookup()`函数查找当前标识符，并标注、返回相应的类型。**如果没有找到，需要报错，并把该表达式类型标注为Object**。

代码实现见代码8.18。报错情况和注意点如下：

- 如果没有找到标识符，报错。
- 关于self，我们不需要进行处理，因为self已经被加入到了SymbolTable中，类型是SELF_TYPE。详见4.19 semant()和问题6.2。

4.11 assign表达式

$$\frac{\begin{array}{l} O(Id) = T \\ O, M, C \vdash e_1 : T' \\ T' \leq T \end{array}}{O, M, C \vdash Id \leftarrow e_1 : T'} \quad [\text{ASSIGN}]$$

首先我们需要检查**被赋值的标识符的类型，即在Symbol Table中lookup**。然后检查右侧表达式的类型。最后，检查两者类型是否相合，使用`isAncestor()`函数进行判断。最后表达式的类型为**右侧表达式的类型**。

代码实现见代码8.19。报错情况和注意点如下：

- 如果标识符类型不存在，应报错类型不存在，标识为Object类型。
- 如果标识符是self，应报错不能赋值给self，标识为SELF_TYPE。
- 如果表达式类型与标识符类型不相合，报错。

4.12 new表达式

$$\frac{T' = \begin{cases} \text{SELF\_TYPE}_C & \text{if } T = \text{SELF\_TYPE} \\ T & \text{otherwise} \end{cases}}{O, M, C \vdash new\ T : T'} \quad [\text{New}]$$

对于new表达式，直接检查其给定的类型T：

- 如果不是SELF_TYPE，则表达式类型就是T。
- 如果是SELF_TYPE，则表达式类型是SELF_TYPE_C。这里，SELF_TYPE_C指**new表达式出现的类中的SELF_TYPE**。

实际在实现时，其实可以直接标注类型T。SELF_TYPE我们会在后面进行相应处理。

代码实现见代码8.20。报错情况和注意点如下：

- 没有找到声明的类型T时，报错，并标注类型Object。

4.13 if表达式

$$\frac{O, M, C \vdash e_1 : Bool \quad O, M, C \vdash e_2 : T_2 \quad O, M, C \vdash e_3 : T_3}{O, M, C \vdash \text{if } e_1 \text{ then } e_2 \text{ else } e_3 \text{ fi} : T_2 \sqcup T_3} \quad [\text{If}]$$

需要依次检查三个子表达式的类型。这里注意返回的结果是**两个子表达式类型T_2和T_3的最小公共祖先类**。获取两个类的最小公共祖先类并不难，只需要在获取两个类的继承路径后，**倒序从Object开始同时遍历。遍历到的最后一个相同类就是两个类的最小公共祖先类**。代码实现见代码8.21。需要注意的是对SELF_TYPE的处理：

- 当两个类都是SELF_TYPE时，应该同样返回SELF_TYPE。（正是因为如此，`getLCA()`函数才应该返回`Symbol`类型：一方面，只需要最小公共祖先类的类型(Symbol)；另一方面，SELF_TYPE并没有对应的Class，从而造成处理上的不方便。）
- 当其中一个类是SELF_TYPE时，需要把SELF_TYPE转换为**当前类**，然后再找最小公共祖先类。

对if表达式检查代码实现见代码8.22。报错情况和注意点如下：

- 如果表达式e_1的类型不是Bool，报错。

4.14 case表达式和branch表达式

$$\frac{\begin{array}{l} O, M, C \vdash e_0 : T_0 \\ O[T_1/x_1], M, C \vdash e_1 : T_1' \\ \vdots \\ O[T_n/x_n], M, C \vdash e_n : T_n' \end{array}}{O, M, C \vdash \text{case } e_0 \text{ of } x_1 : T_1 \Rightarrow e_1; \ldots x_n : T_n \Rightarrow e_n; \text{ esac} : \bigsqcup_{1 \le i \le n} T_i'} \quad [\text{Case}]$$

case语句需要检查里面所有表达式的类型，其结构比较特殊：由一个Expression e_0和一系列Case(也就是分支表达式branch)组成。我们需要关注的重点是**对作用域的处理，因为case语句的每一个branch都会引入新的标识符**。因此，我们需要在每个branch表达式调用`checkType()`时，对作用域进行相应的处理。

首先，我们需要对每个branch进行检查，先创建新作用域，将x_i及其类型加入到作用域，再检查branch的表达式e_i，最后退出作用域。

最后，case表达式的类型是所有branch表达式类型的最小公共祖先类。这只需要依次使用对每两个类

使用`getLCA()`函数即可。

代码实现见代码8.23。报错情况和注意点如下：

- branch的expr的`checkType()`应该在正确的作用域下调用。
- 如果case中含有**声明类型相同的branch**，报错。
- 注意，就算是**声明类型相同的branch**，也要将结果计入到最后的类型中，与规则保持一致。
- branch声明的标识符不能是**self**。关于这点，我发现了一个很神奇的情况，见问题6.2。

4.15 let表达式

$$\frac{\begin{array}{l} T_0' = \begin{cases} \text{SELF\_TYPE}_C & \text{if } T_0 = \text{SELF\_TYPE} \\ T_0 & \text{otherwise} \end{cases} \\ O, M, C \vdash e_1 : T_1 \\ T_1 \leq T_0' \\ O[T_0'/x], M, C \vdash e_2 : T_2 \end{array}}{O, M, C \vdash \text{let } x : T_0 \leftarrow e_1 \text{ in } e_2 : T_2} \quad [\text{Let-Init}]$$

$$\frac{\begin{array}{l} T_0' = \begin{cases} \text{SELF\_TYPE}_C & \text{if } T_0 = \text{SELF\_TYPE} \\ T_0 & \text{otherwise} \end{cases} \\ O[T_0'/x], M, C \vdash e_1 : T_1 \end{array}}{O, M, C \vdash \text{let } x : T_0 \text{ in } e_1 : T_1} \quad [\text{Let-No-Init}]$$

首先，因为let表达式同样也引入了新标识符，所以**我们也需要对作用域进行处理：在表达式开始时创建作用域，加入标识符名字和类型，并在整个let表达式结束时退出作用域**。按照规则，标识符x声明的类型是T_0'，在实际实现时可以**直接使用**T_0。

如果let表达式有初始化的话，还需要检查**初始化类型是否与声明类型相合**。最后，let表达式的类型是**in里面的表达式的类型，这个表达式的类型应该在正确的作用域下进行检查**。

代码实现见代码8.24。报错情况和注意点如下：

- 如果标识符声明类型**未定义**，报错。
- 如果存在初始化表达式，检查与声明类型**是否相合**。不相合则报错。
- 检查标识符是否是self。如果是则报错。
- 这里要注意的一点是，就算**声明类型未定义也要将其加入到scope中**，这是通过测试样例得到的结论。相应的，在检查method时，也需要做相应的处理。详细见问题6.3。

4.16 dispatch表达式

$$\frac{\begin{array}{l} O, M, C \vdash e_0 : T_0 \\ O, M, C \vdash e_1 : T_1 \\ \vdots \\ O, M, C \vdash e_n : T_n \\ T_0' = \begin{cases} C & \text{if } T_0 = \texttt{SELF\_TYPE}_C \\ T_0 & \text{otherwise} \end{cases} \\ M(T_0', f) = (T_1', \ldots, T_n', T_{n+1}') \\ T_i \leq T_i' \quad 1 \leq i \leq n \\ T_{n+1} = \begin{cases} T_0 & \text{if } T_{n+1}' = \texttt{SELF\_TYPE} \\ T_{n+1}' & \text{otherwise} \end{cases} \end{array}}{O, M, C \vdash e_0.f(e_1, \ldots, e_n) : T_{n+1}} \quad [\text{Dispatch}]$$

dispatch表达式的检查是所有规则中最复杂的一个。首先，我们使用调用表达式e_0的`checkType()`函数进行检查。并**对SELF_TYPE的情况做转换（前面都是保留SELF_TYPE，但这里就需要转换了，因为我们需要知道调用哪个类的方法）**。然后，我们**根据方法名找到调用的对应方法（任意一个都可以，因为重载要求返回类型和参数一致）**，并检查：**实参和形参是否一一相合**。最后，表达式类型为函数的返回类型。这里的一个特殊情况是：当函数的返回类型是SELF_TYPE时，dispatch表达式的类型是**调用表达式的类型（注意：是没有经过转换的）**。

代码实现见代码8.25。报错情况和注意点如下：

- 如果**形参和实参个数不匹配**，报错（并不再检查参数类型）。
- 如果某**形参和实参类型不相合**，报错（检查下一个参数）。
- 如果**没有找到对应的方法**（注意继承来的方法也算），报错。
- 注意我们是**先进行的每个实参的类型检查**。这样是为了**保证就算出错无法进行继续检查，也能检查尽可能多的内容。**

4.17 static dispatch表达式

$$\frac{\begin{array}{l} O, M, C \vdash e_0 : T_0 \\ O, M, C \vdash e_1 : T_1 \\ \vdots \\ O, M, C \vdash e_n : T_n \\ T_0 \leq T \\ M(T, f) = (T'_1, \ldots, T'_n, T'_{n+1}) \\ T_i \leq T'_i \quad 1 \leq i \leq n \\ T_{n+1} = \begin{cases} T_0 & \text{if } T'_{n+1} = \texttt{SELF\_TYPE} \\ T'_{n+1} & \text{otherwise} \end{cases} \end{array}}{O, M, C \vdash e_0@T.f(e_1, \ldots, e_n) : T_{n+1}} \quad [\text{StaticDispatch}]$$

static dispatch表达式的类型检查和dispatch表达式的检查基本一致，区别在于：

- 需要检查**调用类是否定义**。
- 需要检查**调用表达式的类型和调用类是否相合**。
- 方法**只检查调用类内的**。

代码实现见代码8.26。报错情况和注意点如下：

- 如果**调用类未定义**，报错，**并停止检查**。
- 如果**调用表达式类型和调用类不相合**，报错，**并停止检查**。
- 其余同dispatch表达式的检查。

4.18 attr和method的Type Checking

$$\frac{\begin{array}{l} O_C(x) = T_0 \\ O_C[\texttt{SELF\_TYPE}_C / self], M, C \vdash e_1 : T_1 \\ T_1 \leq T_0 \end{array}}{O_C, M, C \vdash x : T_0 \leftarrow e_1;} \quad [\text{Attr-Init}]$$

$$\frac{O_C(x) = T}{O_C, M, C \vdash x : T;} \quad [\text{Attr-No-Init}]$$

$$\frac{\begin{array}{l} M(C,f) = (T_1,\ldots,T_n,T_0) \\ O_C[\texttt{SELF\_TYPE}_C/self][T_1/x_1]\ldots[T_n/x_n], M, C \vdash e : T_0' \\ T_0' \leq \begin{cases} \texttt{SELF\_TYPE}_C & \text{if } T_0 = \texttt{SELF\_TYPE} \\ T_0 & \text{otherwise} \end{cases} \end{array}}{O_C, M, C \vdash f(x_1 : T_1, \ldots, x_n : T_n) : T_0 \ \{\ e\ \};} \quad [\text{Method}]$$

cool-manual 12.2 Typing Rules中也给出了属性和方法的类型检查规则。属性和方法本身没有类型，这些规则只是说明需要在属性和方法检查什么。这些检查我们在前面的部分已经完成了。

4.19 program::semant()

至此，静态语义解析的所有内容都已经实现了。现在，我们将前面实现的内容组装在一起。代码见8.27。

检查顺序如下：

- 构建ClassTable。
- 检查类继承性。
- 安装方法和属性，并检查重复。
- 具体检查方法和属性，并检查内部的各表达式。

这里有两点需要注意：

- 如果在构建ClassTable和检查类继承性时就已经出错了，那么就直接停止后面的检查。
- 这里，我将self和它对应的SELF_TYPE加入到了Symbol Table中。这个实现从需求的角度来看没有必要，主要是为了处理4.15 let表达式提到的一种很奇特的情况。见问题6.2。

5. 测试样例

因为我构造的测试样例较多，所以这边**只给出主要的部分**，并附上结果及解析。执行的命令是`./checker [文件名]`，结果**除了inheritTest，均为passed**（inheritTest已给出结果为什么不一致的原因分析），并给出`check_mine.txt`的内容。

5.1 classTest

对Class基本所有错误的测试。包括：

- 对Basic Class和SELF_TYPE的重定义。
- 继承未定义类。
- 继承Int/Bool/Str/SELF_TYPE。
- 重定义类。

代码见代码8.28
结果：测试结果9.1
分析：

- 不能重定义基本类和SELF_TYPE。
- 不能重定义类。
- 不能继承Int/Bool/String/SELF_TYPE。
- 这里我们很明显看到**报错行号是分为两部分递增的，说明语义检查是两遍分开检查的：先检查的类重定义情况，再检查的继承情况。**
- 报错中，没有报C中方法的错误，说明**类检查错误后就会直接终止了。**

5.2 methodTest

对method基本所有错误的测试。包括：

- 重载返回类型不同。
- 重载参数数量不同。
- 重载参数类型不同。
- 重定义**继承的函数。**
- 参数为self。
- 参数重定义。
- 参数类型未定义。
- 返回类型未定义。
- 表达式类型与返回类型不相合。
- 类内重定义方法。

代码见代码8.29
结果：测试结果9.2
分析：

- 我们依旧能看到报错行号被分为了两部分：**首先检查重定义/重载的情况，然后才是其他语义。**这也是**两次遍历AST的一种体现。**
- Class P中的两个方法的报错都是重载非法而非类内重复定义，这说明**检查重定义/重载是从继承类优先开始的。**这与我们之前描述的检查逻辑也是相符的。

- 检查重载时存在优先级：返回类型→参数数量→参数类型。
- self不能作为参数，且不会被加入到作用域中。
- 参数不能重定义。**后者不会覆盖前者**。
- 参数类型未定义时，**仍进行下一项检查，而不是检查下一个参数**。
- 返回类型的检查应该在参数检查之后。

5.3 attrTest

对attribute的基本所有错误测试，包括：

- 属性声明类型未定义。
- 初始化类型与声明类型不相合。
- 重定义属性。
- 重定义继承属性。

代码见代码8.30
结果：测试结果9.3
分析：

- 还是很明显看到检查顺序：**先检查重定义（包括继承），再检查其他语义**。
- Class B中报错是两次继承属性重复，这与我们**检查优先从继承类(从Object到当前类)开始的结论吻合**。
- 尽管声明类型未定义，仍然要检查右侧表达式，这体现**尽可能检查更多内容的原则**。

5.4 exprTest

对Expression的大部分错误检查，基本包含了全部类型的表达式错误。因为错误类型较多，不一一列出，可以详见代码的注释。
代码见代码8.31
结果：测试结果9.4
分析：

- assign出错时，表达式类型是Object。
- new出错时，表达式类型是Object。
- 对于if表达式，就是介词表达式不为Bool类型，计算的表达式类型也是后续两个表达式的最小公共祖先类。
- 这里可以看到，除法表达式中除以0不是**静态语义错误**，本阶段我们不涉及具体值的检查。
- case不能有branch声明相同的类型，但**仍然计算右侧表达式的类型，并计入到最终表达式类型的计算中**。

- let表达式引入的**标识符就算是未定义类型，也会用于后续检查**。可见问题6.3。
- dispatch时，就算调用的方法没找到，也要检查参数的类型（**尽可能检查更多内容！**）。
- static_dispatch时，先要检查调用类型存在性、调用类型和调用表达式类型的相合性。如果有错误，就不进行后面的检查了，**但是参数还是要第一个检查，从报错顺序就能看出来**。

5.5 mainTest

关于main相关的错误和报错优先级等测试。
代码见代码8.32
结果：测试结果9.5
分析：

- 仍然可以看出，**我们是优先检查继承重定义/重载的**，这里Main Class中的两个main方法报的错都是重载错误。
- A只报了一次类内重复定义，说明**类的检查跟程序写的顺序没有关系，是按照继承关系来检查的**。Main Class中报的错说A中main方法的类型是Int也可以看出：**A中第二个main方法相当于不存在**，与我们前面的逻辑推断吻合。
- B中a方法的返回类型错误被检查到了，说明**main检查应该是在第一次AST遍历时进行的**，报错后不停止。
- 如果我们删掉测试用例中的Main Class，我们仍能看到：**B中a方法仍被检查了，且在Main Class未定义错误之后**，这就说明：**Main Class的存在性检查就不属于最开始的Class检查部分，因为Class检查部分如果报错是直接结束检查的**。
- 这里还有一种有趣的情况：如果Main Class中没有main方法，但从其祖先类中继承了main方法，此时**是需要报错main方法不存在的**。

5.6 inheritTest

检查循环继承的情况，包括自己继承自己、祖先类在循环中，但自己不在循环中的情况。
代码见代码8.33
结果：测试结果9.6
分析：

- 上面是expected的结果，下面是我的结果。可以看到，唯一区别在于检查顺序不同，expected的顺序是**严格按照类定义的倒序检查的**，但我这边不行，尽管我已经是按照倒序的逻辑在写的。
- 我推测结果不一样的原因可能是因为**STL map的实现会对键值进行自动排列，导致实际保存的顺序和插入顺序不一致**。进一步探索见问题6.6

5.7 selfTest

测试了self和SELF_TYPE的各种出现情况。包括：

- self作为属性。
- self作为let标识符。
- self作为方法参数。
- self作为case branch标识符。
- 当前类、祖先类、self与SELF_TYPE的各种组合的相合情况。
- 在if语句和case语句中，检查SELF_TYPE参与计算公共祖先的结果。

代码见代码8.34
结果：测试结果9.7
分析：

- self作为属性时，不会被记录为当前类的属性。
- self作为let标识符时，不会加入scope。在其中使用self的类型仍为SELF_TYPE。
- self作为方法参数时，不会加入scope。在其中使用self的类型仍为SELF_TYPE。
- self作为case branch标识符时，**会加入scope从而覆盖掉原来SELF_TYPE的类型**。详见问题6.3。
- 如果是两个SELF_TYPE，则相合。
- SELF_TYPE作为祖先类时，一定不相合。
- SELF_TYPE作为子类时，转换为当前类再检查。
- SELF_TYPE与SELF_TYPE的公共祖先还是SELF_TYPE（不转换）；与其他类的公共祖先需要转换为当前类再计算。
- 可以看到self和SELF_TYPE能正确转换为对应的类型，参考Class A中定义的c方法和Class B中对c方法的调用（注意调用表达式的类型规则）。

5.8 scopeTest

对作用域进行的部分测试，包括所有四种引入作用域的方式及标识符重叠定义的情况。
代码见代码8.35
结果：测试结果9.8（太长，只给出部分）
分析：

- 子类中，可以使用父类中的属性。
- let表达式中，后面的表达式可以使用前面定义的标识符（在出了let表达式后失效），且可以屏蔽之前的定义。
- case表达式中，每个分支可以使用该分支定义的标识符（在出了当前分支后失效），且可以屏蔽之前的定义。

- 方法的参数可以被使用（在出了当前方法后失效），且可以屏蔽之前的定义。

5.9 good

简单构造的一个运行良好的样例，包括嵌套继承、函数重载、使用祖先类的属性和方法、标识符重叠定义、静态调用、self和SELF_TYPE等检查。

代码见代码8.36

结果：测试结果9.9（太长，只给出部分）

分析：能正常解析出正确结果。

6. 问题与思考

6.1 报错(检查)顺序与类型

我第一次写完这个实验的时候，发现一个问题：**官方的报错顺序跟我的不一致**。一开始我忽略了这个问题，但后来就没法忽略了，因为由于报错（也即检查）顺序的不同，导致出现了无法处理的错误。

要解决这个问题，我只能构造了包括但不限于上面谈到的测试用例，并观察了官方的报错结果，推测其检查逻辑。最后对整个检查核心部分进行重构。

根据我的推测，官方的检查顺序应该是：

- 先安装classTable。顺便检查类名是否合法，不合法的不记录。
- 再检查继承关系。如果有问题（包括前面），就直接结束，不检查后面的内容。
- 安装features。这里要检查方法和属性的重名情况（包括继承）。我们可以明显看到，**检查重名和后面检查其他语义属于不同的遍历了**。
- 检查其他语义。

我还大致总结了实现语义检查的两个原则：

- `checkType()`要在合适的地方调用。如果没有必要，就**尽可能晚调用**。
- 要检查**尽可能多**的内容。除了因为信息不足不能再检查的部分，都要检查，尽管可能用不到。

另外，每个检查内部也有一定顺序，如检查重载时，先检查返回类型，再检查参数个数，最后检查参数类型，等。因为内容较多较细，且在前面的部分也有很多叙述了，就不详细展开了。

最后，报错时的表达式类型强调一个`otherwise`，就是**除非真的按照规则给的类型没定义，此时表达式就是Object类型，否则就按照规则给的类型作为表达式类型**。举个例子：

```
    d():Int{
        if 1 then b else c fi
    };
```

```
exprTest.cl:42: Predicate of 'if' does not have type Bool.
exprTest.cl:39: Inferred return type Main of method d does not conform to declared return type Int.
```

这里，b属于B类型，c属于C类型，B和C的最小公共祖先类是Main。尽管if表达式的介词表达式不是Bool类型，但返回的结果仍按照规则是Main，而不是因为出错了就是Object类型。

6.2 一种特殊情况

按照常理，self在任何场合的类型都是SELF_TYPE。然而，在这样一种特殊情况下：

```
    s():Bool{
        case 1 of
            self:Int => self;
        esac
    };
```

在这种情况下，报错的信息居然是**返回类型Int不匹配声明类型Bool**，也就是说，此时**self的类型是声明的Int，尽管已经报了case标识符不能为self的错。**

```
selfTest.cl:81: 'self' cannot be the name of a formal parameter.
selfTest.cl:81: Inferred return type SELF_TYPE of method q does not conform to declared return type Int.
selfTest.cl:86: 'self' cannot be bound in a 'let' expression.
selfTest.cl:85: Inferred return type SELF_TYPE of method r does not conform to declared return type String.
selfTest.cl:91: 'self' bound in 'case'.
selfTest.cl:89: Inferred return type Int of method s does not conform to declared return type Bool.
```

我倾向于这应该是**官方语义解释器的实现失误**。因为同样是self，在let表达式里，比如下面的情况:

```
    r():String{
        let self:Int in self
    };
```

这种情况下的报错信息就是**返回类型SELF_TYPE不匹配声明类型String**，也报了**let标识符不能为self的错**，说明**这种情况下不应该把self加入到SymbolTable中**。其余出现self的情况也是如此。

要修复这个问题也很简单，在case里检测到self时不加入SymbolTable即可，可能就是少写了个else，不然不能解释为什么都报错了还加入到SymbolTable中。我这边还是跟官方的实现保持一致，并为了实

现这种效果，将self也作为变量加入到SymbolTable中（最开始我是针对self作特殊处理，没有加入到SymbolTable中）。

6.3 let表达式类型未定义的情况

最开始，我的`check_method()`中并没有检查方法表达式的类型，因为我觉得，既然都已经调用`checkType()`了，那返回的类型肯定也没问题吧？但是，在下面这个例子中就出现问题了：

```
l():Bool{
    let a:D <- 1 in a
}
```

这里D是一个未定义的类。按理来说，既然未定义，那这个表达式应该就直接返回Object了。但实际的情况是，**这里并没有报返回类型不匹配的错误**。我最开始猜测：**这个表达式的类型会不会是No_type**？因为可能这个表达式错误了，就设置是No_type，这样method在检查返回类型时，就当作没有表达式，略过检查了。但我进行了进一步测试：将这个let表达式作为加法参数与1相加，结果的报错信息是：

```
exprTest.cl:119: Class D of let-bound identifier a is undefined.
exprTest.cl:121: Class D of let-bound identifier self is undefined.
exprTest.cl:121: 'self' cannot be bound in a 'let' expression.
exprTest.cl:123: Inferred type Int of initialization of a does not conform to identifier's declared type String.
exprTest.cl:127: Class D of let-bound identifier a is undefined.
exprTest.cl:129: non-Int arguments: D + Int
exprTest.cl:115: Inferred return type Int of method l does not conform to declared return type Bool.
```

也就是说，这个表达式的类型竟然是没有定义的类型D！为了达到同样的实现，let表达式**无论什么情况下都返回in中表达式的类型，尽管这个类型可能没有定义**。然后在method中添加获得的expr的检查，看这个类型是否存在，不存在的话，就不用进一步检查相合性了。

6.4 报错处理

本次PA已经提供了一个报错方法`semant_error`。最开始我以为所有错的报到curr_class上就可以了，因为只提供了`Class_`参数的报错方法。但在看了官方的报错信息后，我意识到**应该准确的报错误出现对应AST结点的行号**。于是自己实现了`Feature`、`Expression`、`Case`参数的报错方法，其实现是完全一致的。这样，在检查对应内容时就可以直接准确地报错，比如在实现各表达式类型的`checkType()`时，可以直接`semant_error(this)`，表示在当前表达式结点的行号报错。

6.5 遍历了几次AST?

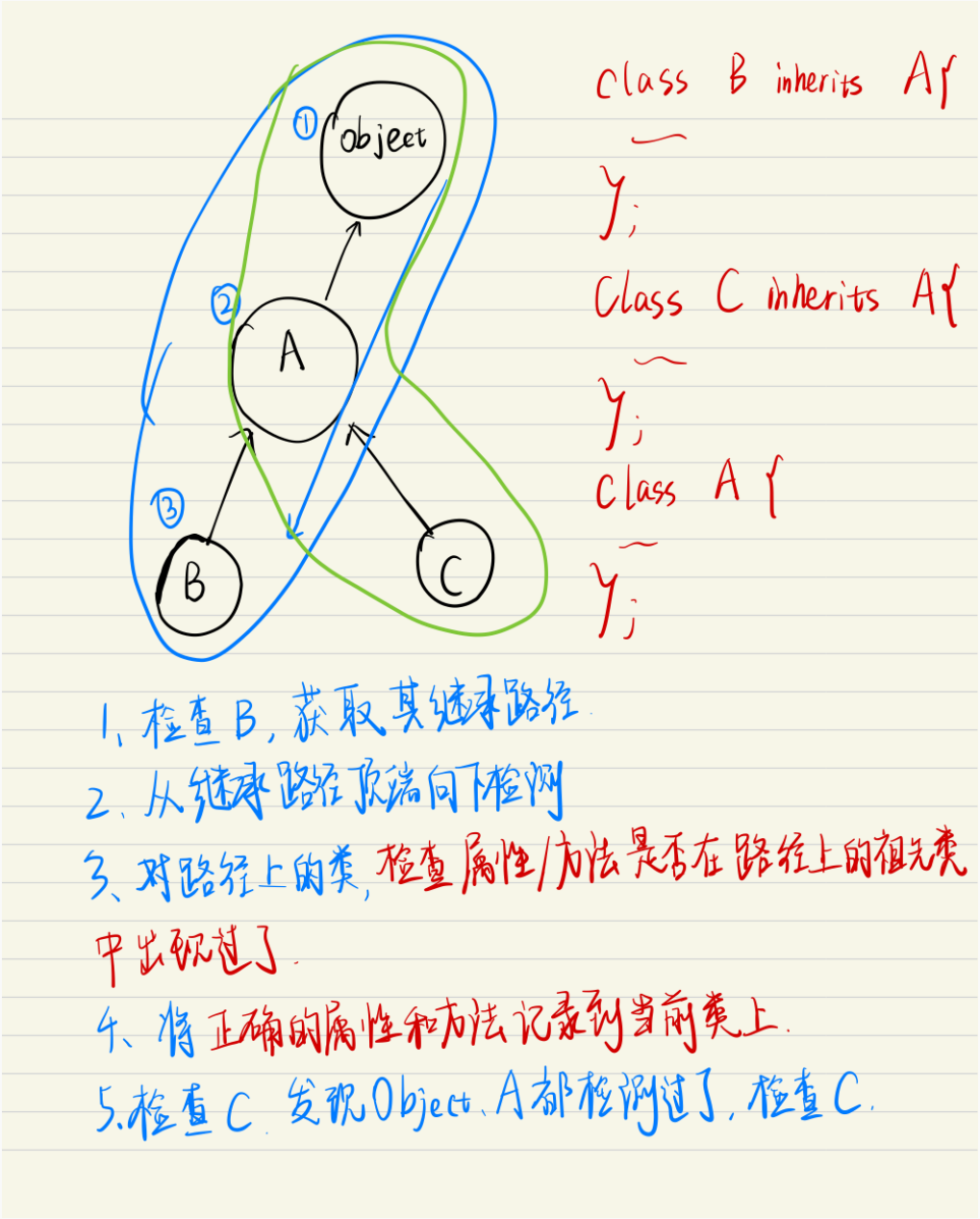
从上面的分析和测试结果中我们可以看到，**大致可以将整个检查过程分为三个阶段：**

- 对类的检查。
- 对方法/属性重复性的检查。
- 其他语义，包括方法/属性类型相合性，表达式类型等检查。

在整个过程中，我们进行了**对AST完整的两次遍历：**

- 第一次，是对方法/属性重复性的检查。我们检查了所有类的所有方法/属性，并把**合法的方法和属性安装到对应的类上**，最终得到一个完整的方法表/属性表。
- 第二次，是对其他语义的检查。在已经有了Class表、方法表、属性表的情况下，检查时就直观简单了很多。

这里我还想强调一下**第一次AST遍历的具体细节**。具体见下图：



具体来讲，检查B时，获取其继承路径，然后：

- 检查Object，看里面的feature是否与**这条路径上Object之上的部分（这里没有）里的feature有重复**。如果有且是错误的，**那么这个feature不会被安装到对应类上**。将正确的feature安装到当前类上。
- 检查A，看里面的feature是否与**这条路径上A之上的部分（这里是Object）里的feature有重复**。因为methodTable和attrTable已经安装好了Object的内容，且**是正确的**，所以可以直接用于检查。
- 检查B...

接下来**按照程序顺序检查C**，同样是获取继承路径。因为Object和A已经检查过了，所以现在直接检查C，并使用它们的feature来判断。

这样的遍历顺序我个人认为有两个好处：

- 检查顺序**不受程序顺序的影响**，每碰到一个类总是先检查**它对应的这条路径上的所有类**。
- 因为是**自顶向下的检查**，所以**祖先类已经安装好的内容可以用于子类feature的检查**。这样的检查顺序也保证了我们为每个类记录的feature**一定是正确的**。

6.6 检查继承的顺序

在循环继承测试中，我发现我的结果与官方实现不一致：官方的报错顺序**严格与程序定义顺序相反**。

我推测报错顺序不一致的原因是**STL map实现会将作为键值的Symbol自动排序**，从而**做不到按照插入顺序的倒序进行遍历**。经过查阅资料，我发现**STL map可以自定义排序函数，不对键值进行排序**。但我尝试了一段时间也没有得到期望的结果，所以我就放弃这部分的修改了，因为这个报错我个人认为影响不是很大。

7. 总结与感想

本次实验的代码量和难度都十分大，远超前面两次实验。为了把结果做到尽可能好，我代码重写的次数就不少于三次，主要难点在于：

- 本身**需要检查、需要实现的内容非常多**。
- 报错情况**非常多**，且有的报错**非常刁钻**，很难简单想到。
- 需要与官方的结果进行对比，推测它的实现逻辑，并思考：**为什么要这样实现？比起我的实现优势在哪里？**

除了实现的繁琐性外，本身理解上的难度也不低：初次接触时我很难理解符号表这一存在，尤其是在类继承时的符号表的表现，也是在实验过程中花费大量时间逐渐理解的。

但从收获上来讲，本次实验带来的不仅是对编译全过程的更深理解，也是对测试能力、优化能力的很好训练。在第一次写完初版代码进行测试的时候，当我看到报的错误好像“只是顺序不一样”时，我也有“就这么算了”的心理。但随着后来更进一步的测试，我逐渐理解了官方的检测逻辑，并意识到这件事我

花时间是可以优化的，于是我对自己的代码进行了不断的优化、重写，并在这个过程中也发现了之前自以为详尽的测试中许多没发现的刁钻问题。尽管耗费了极多的时间，这样的成就感是难以言喻的。

8. 代码

8.1 构建ClassTable

```
void ClassTable::install_basic_classes()
{
    ...
    // add basic classes
    classTable[Object] = Object_class;
    classTable[IO] = IO_class;
    classTable[Int] = Int_class;
    classTable[Bool] = Bool_class;
    classTable[Str] = Str_class;
}

// install all classes
ClassTable::ClassTable(Classes classes) : semant_errors(0),
error_stream(cerr)
{
    install_basic_classes();

    for (int i = classes->first(); classes->more(i);
        i = classes->next(i))
    {
        curr_class = classes->nth(i);
        Symbol cname = curr_class->get_name();
        Symbol pname = curr_class->get_parentname();

        // 1. class name can't be Int/Str/Bool/IO/Object/SELF_TYPE
        if (cname == Object || cname == IO || cname == Int
        || cname == Str || cname == Bool || cname == SELF_TYPE)
        {
            semant_error(curr_class)
            << "Redefinition of basic class " << cname << ".\n";
            ifStop = true;
        }

        // 2. class name shouldn't be redefined
        else if (classTable.find(cname) != classTable.end())
```

```
        {
            semant_error(curr_class) << "Class " << cname
            << " was previously defined.\n";
            ifStop = true;
        }
        else
            classTable[cname] = curr_class;
    }
}
```

8.2 对继承的检查

```
void ClassTable::check_inheritance()
{
    for (auto it = classTable.rbegin(); it != classTable.rend(); ++it)
    {
        Symbol parent_name = it->second->get_parentname();
        // parnet class cannot be Int/Bool/Str/SELF_TYPE
        if (parent_name == Int || parent_name == Bool
            || parent_name == Str || parent_name == SELF_TYPE)
        {
            semant_error(it->second) << "Class " << it->first
            << " cannot inherit class " << parent_name << ".\n";
            ifStop = true;
        }
        // note that Object's parent class is no_class
        // parent class should be defined
        else if (parent_name != No_class
        && classTable.find(parent_name) == classTable.end())
        {
            semant_error(it->second) << "Class " << it->first
            << " inherits from an undefined class "
            << parent_name << ".\n";
            ifStop = true;
        }
    }

    if (ifStop)
        return;

    for (auto it = classTable.rbegin(); it != classTable.rend(); ++it)
    {
        curr_class = it->second;
        Symbol cname = it->first;

        if (cname == Object)
            continue;
        // maintain slow and fast for each class
        Symbol slow = it->first;
        Symbol fast = it->first;
        while (true)
```

```
        {
            slow = classTable[slow]->get_parentname();
            fast = classTable[fast]->get_parentname();
            if (fast == Object)
                break;
            fast = classTable[fast]->get_parentname();
            if (fast == Object)
                break;
            // there exists a circle
            if (slow == fast)
            {
                semant_error(curr_class) << "Class " << cname
                << ", or an ancestor of " << cname
                << ", is involved in an inheritance cycle.\n";
                ifStop = true; // need to stop
                break;
            }
        }
    }
}
```

8.3 获取继承路径

```
std::vector<Class_> ClassTable::getInheritancePath(Class_ c)
{
    std::vector<Class_> path;
    Class_ cls = c;

    while (true)
    {
        path.push_back(cls);
        if (cls->get_name() == Object)
            break;
        cls = classTable[cls->get_parentname()];
    }
    return path;
}

std::vector<Class_> ClassTable::getInheritancePath(Symbol s)
{
   return getInheritancePath(classTable[s]);
}
```

8.4 构建methodTable和attrTable，并检查

```
void ClassTable::install_attrs_and_methods()
{
    // check as the program sequence?
    std::map<Class_, bool> checked;
    for (auto it = classTable.begin(); it != classTable.end(); ++it)
    {
        if (checked[it->second])
            continue;
        curr_class = it->second;
        auto path = getInheritancePath(curr_class);

        // check all class on the path,from Object to curr_class
        // size_t problem :(
        for (int pi = path.size() - 1; pi >= 0; --pi)
        {
            if (checked[path[pi]])
                continue;
            // checked
            checked[path[pi]] = true;
            std::vector<method_class *> curr_methods;
            std::vector<attr_class *> curr_attrs;
            auto features = path[pi]->get_features();
            for (int i = features->first(); features->more(i);
            i = features->next(i))
            {
                auto feature = features->nth(i);
                if (feature->is_attr())
                {
                    auto attr = (attr_class *)feature;
                    // self check
                    if (attr->get_name() == self)
                    {
                        semant_error(attr)
                        << "'self' cannot be the name of an attribute.\n";
                        continue;
                    }
                    // find in inherited class
                    bool repeat = false;
                    // now checking class's ancestors(start from Object)
```

```
                for (int pj = path.size() - 1; pj > pi; --pj)
                {
                    // this parent class's attrs
                    // (after checking.Error attrs should not be installed)
                    auto pattrs = attrTable[path[pj]];
                    for (auto ait = pattrs.begin(); ait != pattrs.end(); ++ait)
                    {
                        if ((*ait)->get_name() == attr->get_name())
                        {
                            semant_error(attr) << "Attribute "
                            << attr->get_name()
                            << " is an attribute of an inherited class.\n";
                            repeat = true;
                            break;
                        }
                    }
                    if (repeat)
                        break;
                }
                // find in curr class
                if (!repeat)
                {
                    for (auto ait = curr_attrs.begin(); ait != curr_attrs.end();
                    ++ait)
                    {
                        if ((*ait)->get_name() == attr->get_name())
                        {
                            semant_error(attr) << "Attribute "
                            << attr->get_name()
                            << " is multiply defined in class.\n";
                            repeat = true;
                            break;
                        }
                    }
                }

                if (!repeat)
                    curr_attrs.push_back(attr);
            }
            else
            {
                auto method = (method_class *)feature;
```

```
                // find in inherited class
                bool repeat = false;
                for (int pj = path.size() - 1; pj > pi; --pj)
                {
                    auto pmethods = methodTable[path[pj]];
                    for (auto mit = pmethods.begin(); mit != pmethods.end();
                    ++mit)
                    {
                        // check sequence is no matter
                        if ((*mit)->get_name() == method->get_name())
                        {
                            if (!check_override(method, path))
                                repeat = true;
                            break;
                        }
                    }
                    if (repeat)
                        break;
                }
                // find in curr class
                if (!repeat)
                {
                    for (auto mit = curr_methods.begin();
                    mit != curr_methods.end(); ++mit)
                    {
                        if ((*mit)->get_name() == method->get_name())
                        {
                            semant_error(method) << "Method "
                            << method->get_name()
                            << " is multiply defined.\n";
                            repeat = true;
                            break;
                        }
                    }
                }

                if (!repeat)
                    curr_methods.push_back(method);
            }
        }

        methodTable[path[pi]] = curr_methods;
```

```
            attrTable[path[pi]] = curr_attrs;
        }
    }
    // check main
    if(mainExist)
    {
        auto methods = methodTable[classTable[Main]];
        for(auto it = methods.begin();it != methods.end();++it)
        {
            if((*it)->get_name() == main_meth)
            {
                methodExist = true;
                auto main_method = *it;
                auto formals = main_method->get_formals();
                if(formals->len() == 0)
                    methodFormal = true;
                break;
            }
        }
    }

    if(!mainExist)
        semant_error() << "Class Main is not defined.\n";
    else if(!methodExist)
        semant_error(classTable[Main]) << "No 'main' method in class Main.\n";
    else if(!methodFormal)
        semant_error(classTable[Main])
        << "'main' method in class Main should have no arguments.\n";
}
```

8.5 填充Class的作用域

```
void ClassTable::check_attrs_and_methods()
{
    // we check every class
    for (auto it = classTable.begin(); it != classTable.end(); ++it)
    {
        if (semant_debug)
            cerr << "check every class: " << it->first << "\n";
        curr_class = it->second;
        // enter current scope
        objectEnvironment.enterscope();
        // get the inheritance path
        auto curr_path = getInheritancePath(curr_class);

        // prepare scope:attrs
        for (auto pit = curr_path.rbegin(); pit != curr_path.rend(); ++pit)
        {
            auto attrs = attrTable[*pit];
            for (auto ait = attrs.begin(); ait != attrs.end(); ++ait)
            {
                auto attr = *ait;
                objectEnvironment.addid(attr->get_name(),
                new Symbol(attr->get_type_decl()));
            }
        }

        // check curr_class's features
        // notice we'd better check in sequence
        auto features = curr_class->get_features();
        for (int i = features->first(); features->more(i); i = features->next(i))
        {
            auto feature = features->nth(i);
            if (feature->is_attr())
                check_attr((attr_class *)feature);
            else
                check_method((method_class *)feature);
        }

        // exit current class's scope
        objectEnvironment.exitscope();
```

```
    }
}
```

8.6 检查属性类型是否定义

```
void ClassTable::check_attr(attr_class *attr)
{
    Symbol attr_name = attr->get_name();
    Symbol attr_type = attr->get_type_decl();
    bool flag = false;

    // handle SELF_TYPE
    // (this will not change the real attr type, just used to do checking)
    if (attr_type == SELF_TYPE)
        attr_type = curr_class->get_name();

    // 1. does type_decl exist?
    // we should notice type prim_slot. It is the value type of Int and Bool
    if (attr_type != prim_slot && classTable.find(attr_type) == classTable.end())
    {
        semant_error(attr) << "Class " << attr_type << " of attribute "
        << attr_name << " is undefined.\n";
        flag = true;
    }
    // 2. does init_type is conformed to type_decl?
    // note we must handle no_type! very important!
    Symbol attr_init = attr->get_init()->checkType();
    if (attr_init == SELF_TYPE)
        attr_init = curr_class->get_name();

    if (!flag && attr_init != No_type && !isAncestor(attr_type, attr_init))
        semant_error(attr) << "Inferred type " << attr_init
        << " of initialization of attribute " << attr_name
        << " does not conform to declared type " << attr_type << ".\n";
}
```

8.7 检查相合性

```
// check if A is B's ancestor?(i.e. B is conformed to A )
bool ClassTable::isAncestor(Class_ A, Class_ B)
{
    std::vector<Class_> pathB = getInheritancePath(B);

    for (auto it = pathB.begin(); it != pathB.end(); ++it)
    {
        if ((*it)->get_name() == A->get_name())
            return true;
    }
    return false;
}

bool ClassTable::isAncestor(Symbol A, Symbol B)
{
    if (A == SELF_TYPE && B == SELF_TYPE)
        return true;
    else if (A == SELF_TYPE)
        return false;
    else if (B == SELF_TYPE)
        return isAncestor(classTable[A], curr_class);
    return isAncestor(classTable[A], classTable[B]);
}
```

8.8 检查方法重载

```
bool ClassTable::check_override(method_class *method,
const std::vector<Class_> &path)
{
    // from curr_class's parent to object,check if exists this method
    for (auto it = path.begin() + 1; it != path.end(); ++it)
    {
        // parent methods
        auto methods = methodTable[*it];
        // check if exist same method
        for (auto mit = methods.begin(); mit != methods.end(); ++mit)
        {
            if (semant_debug)
                cerr << "in Class " << (*it)->get_name() << ":checking method "
                 << (*mit)->get_name() << ".\n";
            method_class *parent_method = *mit;
            if (parent_method->get_name() == method->get_name())
            {
                // check:
                // 1. is return type the same?
                if (parent_method->get_return_type() != method->get_return_type())
                {
                    semant_error(method) << "In redefined method "
                    << method->get_name() << ", return type "
                    << method->get_return_type()
                    << " is different from original return type "
                    << parent_method->get_return_type() << ".\n";

                    return false;
                }

                // get the formals
                Formals parent_formals = parent_method->get_formals();
                Formals formals = method->get_formals();

                // 2. is formal num the same?
                if (parent_formals->len() != formals->len())
                {
                    semant_error(method)
                    << "Incompatible number of formal parameters in\
```

```
                    redefined method " << method->get_name() << ".\n";
                return false;
            }

            // 3. is formal type the same? (sequence must be the same)
            for (int i = formals->first(); formals->more(i);
            i = formals->next(i))
            {
                auto formal = formals->nth(i);
                auto parent_formal = parent_formals->nth(i);

                if (formal->get_type_decl() != parent_formal->get_type_decl())
                {
                    semant_error(method) << "In redefined method "
                    << method->get_name() << ", parameter type "
                    << formal->get_type_decl()
                    << " is different from original type "
                    << parent_formal->get_type_decl() << "\n";
                    // why doesn't this error end with a period :)?
                    return false;
                }
            }
        }
    }
    return true;
}
```

8.9 参数检查

```
void ClassTable::check_formals(method_class *method, Formals formals)
{
    std::set<Symbol> formal_set;
    for (int i = formals->first(); formals->more(i); i = formals->next(i))
    {
        Formal curr_formal = formals->nth(i);

        // 1. is type "SELF_TYPE"?
        if (curr_formal->get_type_decl() == SELF_TYPE)
            semant_error(method) << "Formal parameter " << curr_formal->get_name()
            << " cannot have type SELF_TYPE.\n";

        // 2. is formal declared type defined?
        // note we continue to check next step
        else if (classTable.find(curr_formal->get_type_decl()) == classTable.end())
            semant_error(method) << "Class " << curr_formal->get_type_decl()
            << " of formal parameter " << curr_formal->get_name()
            << " is undefined.\n";

        // 3. is name "self"?
        if (curr_formal->get_name() == self)
        {
            semant_error(method)
            << "'self' cannot be the name of a formal parameter.\n";
            continue;
        }

        // 4. is formal redefined?
        if (formal_set.count(curr_formal->get_name()))
        {
            semant_error(method) << "Formal parameter " << curr_formal->get_name()
            << " is multiply defined.\n";
            continue;
        }
        else
            formal_set.insert(curr_formal->get_name());

        // and we need to add formal to current scope
        objectEnvironment.addid(curr_formal->get_name(),
```

```
            new Symbol(curr_formal->get_type_decl()));
        }
    }
```

8.10 检查返回类型(包括其他检查的调用)

```
void ClassTable::check_method(method_class *method)
{
    // note that we need to enter a new scope for a method
    objectEnvironment.enterscope();
    Symbol method_name = method->get_name();
    Symbol method_returnType = method->get_return_type();

    Formals method_formals = method->get_formals();

    // 1. check the formals
    // note the sequence.We must check each formal first,
    // to add them to the current scope,then check the expr type
    check_formals(method, method_formals);

    // 2. is return_type defined?
    // if undefined, we should return at once. Otherwise checking the expr type
    // and the return type will cause problem
    // notice SELF_TYPE. we should not convert in method definition!
    bool flag = false;
    if (method_returnType != SELF_TYPE
        && classTable.find(method_returnType) == classTable.end())
    {
        semant_error(method) << "Undefined return type " << method_returnType
        << " in method " << method_name << ".\n";
        flag = true;
    }

    Symbol method_exprType = method->get_expr()->checkType();
    // 3. check expr_type
    if (method_exprType != SELF_TYPE
        && classTable.find(method_exprType) == classTable.end())
        flag = true;

    // 4. is expr_type is conformed to return_type?
    if (!flag && method_exprType != No_type
        && !isAncestor(method_returnType, method_exprType))
        semant_error(method) << "Inferred return type " << method_exprType
        << " of method " << method_name
        << " does not conform to declared return type " << method_returnType << ".\n";
```

```
    // exit this scope
    objectEnvironment.exitscope();
}
```

8.11 常量表达式检查

```
Symbol int_const_class::checkType()
{
    type = Int;
    return type;
}

Symbol bool_const_class::checkType()
{
    type = Bool;
    return type;
}

Symbol string_const_class::checkType()
{
    type = Str;
    return type;
}

Symbol no_expr_class::checkType()
{
    type = No_type;
    return type;
}
```

8.12 算数/比较表达式检查(以加法为例)

```
Symbol plus_class::checkType()
{
    Symbol e1_type = e1->checkType();
    Symbol e2_type = e2->checkType();

    if (e1_type != Int || e2_type != Int)
        classtable->semant_error(this) << "non-Int arguments: "
        << e1_type << " + " << e2_type << "\n";

    type = Int;
    return type;
}
```

8.13 逻辑/算数取反表达式检查

```
Symbol neg_class::checkType()
{
    Symbol e1_type = e1->checkType();

    if (e1_type != Int)
        classtable->semant_error(this) << "Argument of '~' has type "
        << e1_type << " instead of Int.\n";

    type = Int;
    return type;
}

Symbol comp_class::checkType()
{
    Symbol e1_type = e1->checkType();

    if (e1_type != Bool)
        classtable->semant_error(this) << "Argument of 'not' has type "
        << e1_type << " instead of Bool.\n";

    type = Bool;
    return type;
}
```

8.14 isvoid表达式检查

```
Symbol isvoid_class::checkType()
{
    e1->checkType();
    type = Bool;
    return type;
}
```

8.15 等于表达式检查

```
Symbol eq_class::checkType()
{
    Symbol e1_type = e1->checkType();
    Symbol e2_type = e2->checkType();

    if (e1_type == Int || e1_type == Str || e1_type == Bool
        || e2_type == Int || e2_type == Str || e2_type == Bool)
    {
        if (e1_type != e2_type)
            classtable->semant_error(this)
            << "Illegal comparison with a basic type.\n";
    }

    type = Bool;
    return type;
}
```

8.16 循环表达式检查

```
Symbol loop_class::checkType()
{
    Symbol pred_type = pred->checkType();

    if (pred_type != Bool)
        classtable->semant_error(this) << "Loop condition does not have type Bool.\n";

    Symbol body_type = body->checkType();
    type = Object;
    return type;
}
```

8.17 block表达式检查

```
Symbol block_class::checkType()
{
    for (int i = body->first(); body->more(i); i = body->next(i))
        type = body->nth(i)->checkType();
    return type;
}
```

8.18 object表达式检查

```
Symbol object_class::checkType()
{
    // handle self
    if (name == self)
    {
        type = SELF_TYPE;
        return type;
    }

    // lookup from scope list
    auto temp_type = classtable->objectEnvironment.lookup(name);
    if (temp_type == NULL)
    {
        classtable->semant_error(this) << "Undeclared identifier " << name << ".\n";
        type = Object;
    }
    else
        type = *temp_type;

    return type;
}
```

8.19 assign表达式检查

```
Symbol assign_class::checkType()
{
    auto temp_type = classtable->objectEnvironment.lookup(name);
    Symbol id_type;
    // check LHS
    if (name == self)
    {
        classtable->semant_error(this) << "Cannot assign to 'self'.\n";
        id_type = SELF_TYPE;
    }
    else if (temp_type == NULL)
    {
        classtable->semant_error(this) << "Assignment to undeclared variable "
        << name << ".\n";
        id_type = Object;
    }
    else
        id_type = *temp_type;

    // check RHS
    Symbol e_type = expr->checkType();

    if (!classtable->isAncestor(id_type, e_type))
        classtable->semant_error(this) << "Type " << e_type
        << " of assigned expression does not conform to declared type "
        << id_type << " of identifier " << name << ".\n";

    // type is RHS
    type = e_type;

    return type;
}
```

8.20 new表达式检查

```
Symbol new__class::checkType()
{
    // check whether the type is already declared
    if (type_name != SELF_TYPE
        && classtable->classTable.find(type_name) == classtable->classTable.end())
    {
        classtable->semant_error(this) << "'new' used with undefined class "
        << type_name << ".\n";
        type = Object;
        return type;
    }
    type = type_name;
    return type;
}
```

8.21 获取最小公共祖先类

```
Symbol ClassTable::getLCA(Class_ A, Class_ B)
{
    std::vector<Class_> pathA = getInheritancePath(A);
    std::vector<Class_> pathB = getInheritancePath(B);

    Class_ LCA = classTable[Object];

    for (auto itA = pathA.rbegin(), itB = pathB.rbegin(); itA != pathA.rend()
        && itB != pathB.rend(); ++itA, ++itB)
    {
        if ((*itA)->get_name() == (*itB)->get_name())
            LCA = *itA;
    }

    return LCA->get_name();
}

Symbol ClassTable::getLCA(Symbol A, Symbol B)
{
    // handle SELF_TYPE
    if (A == SELF_TYPE && B == SELF_TYPE)
        return SELF_TYPE;
    else if (A == SELF_TYPE)
        return getLCA(curr_class, classTable[B]);
    else if (B == SELF_TYPE)
        return getLCA(classTable[A], curr_class);

    return getLCA(classTable[A], classTable[B]);
}
```

8.22 if表达式检查

```
Symbol cond_class::checkType()
{
    Symbol pred_type = pred->checkType();

    if (pred_type != Bool)
        classtable->semant_error(this) << "Predicate of 'if' does not have type Bool.\n";

    Symbol then_type = then_exp->checkType();
    Symbol else_type = else_exp->checkType();

    type = classtable->getLCA(then_type, else_type)->get_name();
    return type;
}
```

8.23 case表达式和branch表达式的检查

```
Symbol branch_class::checkType()
{
    // note we need to enter a new scope each branch
    classtable->objectEnvironment.enterscope();
    if (name == self)
        classtable->semant_error(this) << "'self' bound in 'case'.\n";
    classtable->objectEnvironment.addid(name, new Symbol(type_decl));
    // we should check type in this scope
    Symbol type = expr->checkType();
    classtable->objectEnvironment.exitscope();
    return type;
}

Symbol typcase_class::checkType()
{
    expr->checkType();

    // use a set to check if duplicate
    std::set<Symbol> case_types;

    for (int i = cases->first(); cases->more(i); i = cases->next(i))
    {
        branch_class *branch = (branch_class *)cases->nth(i);

        // check if duplicate(note we need to check type,not name)
        if (case_types.count(branch->get_type_decl()))
            // notice here to report branch's line number
            classtable->semant_error(branch) << "Duplicate branch "
            << branch->get_type_decl() << " in case statement.\n";
        else
            case_types.insert(branch->get_type_decl());

        // we find that branch_type is computed afterward
        Symbol branch_type = branch->checkType();

        // even the branch is duplicate, its branch type should still be gathered
        if (i > 0)
            type = classtable->getLCA(type, branch_type);
        else
```

```
            type = branch_type;
        }

        return type;
    }
```

8.24 let表达式检查

```
Symbol let_class::checkType()
{
    Symbol init_type = init->checkType();
    bool flag = true;

    // check if type_decl is defined
    // notice SELF_TYPE
    if (type_decl != SELF_TYPE
        && classtable->classTable.find(type_decl) == classtable->classTable.end())
    {
        classtable->semant_error(this) << "Class " << type_decl
        << " of let-bound identifier " << identifier << " is undefined.\n";
        flag = false;
    }

    // check if conformed
    if (flag && init_type != No_type)
    {
        if (!classtable->isAncestor(type_decl, init_type))
            classtable->semant_error(this) << "Inferred type " << init_type
            << " of initialization of "<< identifier
            << " does not conform to identifier's declared type "
            << type_decl << ".\n";
    }

    // enter a new scope
    classtable->objectEnvironment.enterscope();
    // identifier type is type_decl
    if (identifier == self)
        classtable->semant_error(this)
        << "'self' cannot be bound in a 'let' expression.\n";
    else
        classtable->objectEnvironment.addid(identifier, new Symbol(type_decl));
    // check body_type in current scope
    Symbol body_type = body->checkType();
    classtable->objectEnvironment.exitscope();

    type = body_type;
```

```
    return type;
}
```

8.25 dispatch表达式检查

```
Symbol dispatch_class::checkType()
{
    bool ifFind = false;
    Symbol e_type = expr->checkType();
    Symbol e_type_0 = e_type;
    std::vector<Symbol> actual_types;
    for(int i = actual->first();actual->more(i);i=actual->next(i))
        actual_types.push_back(actual->nth(i)->checkType());
    // convert SELF_TYPE
    if (e_type == SELF_TYPE)
        e_type = classtable->curr_class->get_name();

    // we should find the method f
    auto path = classtable->getInheritancePath(e_type);
    for (auto pit = path.begin(); pit != path.end(); ++pit)
    {
        auto methods = classtable->methodTable[*pit];
        for (auto mit = methods.begin(); mit != methods.end(); ++mit)
        {
            auto curr_method = *mit;
            // find(only the first)
            if (curr_method->get_name() == name)
            {
                ifFind = true;

                // check the formals
                Formals method_formals = curr_method->get_formals();

                // check if actual num is the same as formal num
                if (method_formals->len() != actual->len())
                    classtable->semant_error(this) << "Method " << name
                    << " called with wrong number of arguments.\n";

                // check each formal
                else
                    for (int i = method_formals->first(); method_formals->more(i);
                        i = method_formals->next(i))
                    {
                        Formal curr_formal = method_formals->nth(i);
```

```
                        Symbol formal_type = curr_formal->get_type_decl();

                        // check if actual type is conformed to formal_type
                        Symbol actual_type = actual_types[i];
                        if (!classtable->isAncestor(formal_type, actual_type))
                            classtable->semant_error(this) << "In call of method "
                            << name << ", type " << actual_type << " of parameter "
                            << curr_formal->get_name()
                            << " does not conform to declared type "
                            << formal_type << ".\n";
                    }

                type = curr_method->get_return_type();
                break;
            }
        }
        if (ifFind)
            break;
    }

    if (ifFind)
    {
        if (type == SELF_TYPE)
            type = e_type_0; // should return original type
        return type;
    }
    else
    {
        classtable->semant_error(this) << "Dispatch to undefined method "
        << name << ".\n";
        type = Object;
        return type;
    }
}
```

8.26 static dispatch表达式检查

```
Symbol static_dispatch_class::checkType()
{
    bool ifFind = false;
    Symbol e_type = expr->checkType();
    std::vector<Symbol> actual_types;
    for(int i = actual->first();actual->more(i);i=actual->next(i))
        actual_types.push_back(actual->nth(i)->checkType());

    // check if type_name is defined
    if (classtable->classTable.find(type_name) == classtable->classTable.end())
    {
        classtable->semant_error(this) << "Static dispatch to undefined class "
        << type_name << ".\n";
        type = Object;
        return type;
    }

    // check if e_type is conformed to type_name
    if (!classtable->isAncestor(type_name, e_type))
    {
        classtable->semant_error(this) << "Expression type " << e_type
        << " does not conform to declared static dispatch type "
        << type_name << ".\n";
        type = Object;
        return type;
    }

    // get methods
    auto methods = classtable->methodTable[classtable->classTable[type_name]];
    // find corresponding method
    for (auto mit = methods.begin(); mit != methods.end(); ++mit)
    {
        auto curr_method = *mit;
        if (curr_method->get_name() == name)
        {
            ifFind = true;

            // check the formals
            Formals method_formals = curr_method->get_formals();
```

```
        // check if actual num is the same as formal num
        if (method_formals->len() != actual->len())
            classtable->semant_error(this) << "Method " << name
            << " invoked with wrong number of arguments.\n";

        // check each formal
        else
            for (int i = method_formals->first(); method_formals->more(i);
            i = method_formals->next(i))
            {
                Formal curr_formal = method_formals->nth(i);
                Symbol formal_type = curr_formal->get_type_decl();

                // check if actual type is conformed to formal_type
                Symbol actual_type = actual_types[i];
                if (!classtable->isAncestor(formal_type, actual_type))

                    classtable->semant_error(this) << "In call of method "
                    << name  << ", type " << actual_type
                    << " of parameter " << curr_formal->get_name()
                    << " does not conform to declared type "
                    << formal_type << ".\n";
            }

        type = curr_method->get_return_type();
        break;
    }
}

if (ifFind)
{
    if (type == SELF_TYPE)
        type = e_type;
    return type;
}
else
{
    classtable->semant_error(this) << "Static dispatch to undefined method "
    << name << ".\n";
    type = Object;
    return type;
```

```
    }
}
```

8.27 program::semant()

```
void program_class::semant()
{
    initialize_constants();

    /* ClassTable constructor may do some semantic analysis */
    classtable = new ClassTable(classes);
    // install self
    classtable->objectEnvironment.enterscope();
    classtable->objectEnvironment.addid(self, new Symbol(SELF_TYPE));

    /* some semantic analysis code may go here */
    // check inheritance first
    classtable->check_inheritance();

    if (!ifStop)
    {
        classtable->install_attrs_and_methods();
        classtable->check_attrs_and_methods();
    }

    if (classtable->errors())
    {
        cerr << "Compilation halted due to static semantic errors." << endl;
        exit(1);
    }
}
```

8.28 测试：classTest

```
// inherits from an undefined class
Class A inherits B{

};

// inherits from Int/Bool/Str/SELF_TYPE
Class C inherits SELF_TYPE{

};

// class name is SELF_TYPE
Class SELF_TYPE{

};

// redefintion
Class C{
    func():INT{
        1
    };
};

// redefinition of basic class and SELF_TYPE
Class Int{

};

Class Int{

};

Class String{

};

Class Bool{

};
```

```
Class IO{

};

Class Object{

};

Class SELF_TYPE{

};
```

8.29 测试：methodTest

```
class Main{
    main():Int{
        1
    };
};

class A{
    func(a:Int,b:Bool):Int{
        1
    };
};

// invalid overrides
// first check return type!
Class B inherits A{
    func(a:Int,b:String):String{
        "abc"
    };
};

// if you put a 'self'
Class C inherits A{
    func(a:Int,self:String):String{
        "abc"
    };
};

// then func num
Class D inherits A{
    func(a:Int,b:String,c:Int):Int{
        1
    };
};

// finally formal type
Class E inherits A{
    func(b:Bool,a:Int):Int{
        1
    };
```

```
};

// check redefinition for inheritance
Class P inherits A{
    func():Int{
        1
    };
    func():Int{
        1
    };
};


class F{
    //  method has a formal name 'self'
    //  notice won't show redefinition error
    a(self:Int,self:Int):Int{
        self
    };
    //  SELF_TYPE
    f(self:SELF_TYPE):Int{
        1
    };
    //  formal redefinition
    b(b:Int,b:Bool):Int{
        b
    };
    //  formal type is undefined
    //  we find that when decl_type undefined, go on next check(i.e. is it self?), not check ne
    c(x:INT,x:INT,self:STR):Int{
        1
    };
    //  return type is undefined
    //  indicates that return type is checked after formal checks
    d(self:Int,self:INT):INT{
        1
    };

    //  expr type does not conform to return type
    e(self:Int,self:INT):Int{
        "abc"
    };
```

```
    //  redefinition(Cool does not support override in one class)
    e():Int{
        1
    };
};
```

8.30 测试：attrTest

```
Class Main{
    main():Int{
        1
    };
};

Class A{
    // decl_type undefined
    a:INT <- c;
    // not conform
    b:Int <- "abc";
    // redefinition
    b:String;
};

Class B inherits A{
    // redefinition attr for inheritance
    a:Int;
    a:Int;
};
```

8.31 测试：exprTest

```
Class Main{
    main():Int {
        1
    };
};

Class B inherits Main{

};

Class C inherits Main{

};

Class Z{
    func(a:Int,b:Int):Int{
        1
    };
};

Class A inherits Z{
    b:B <- new B;
    c:C <- new C;
    // assign: undefined id(lhs and rhs)
    a():Object{
        a <- a
    };
    b():Object{
        // assign: not conform
        b <- "ab"
    };

    c():Int{
        // new : undefined class
        // should be type Object
        new C
    };

    d():Int{
```

```
        // if: pred is not Bool.
        // here we check the return type is not Object,
        // though pred is wrong
        if 1 then b else c fi
    };

    e():Int{
        // arith: incorrect type of LHS or RHS
        // (block expr cannot be semantically wrong)
        {
            1 + "a";
            "a" - 1;
            "a" * true;
            "a" / 0;        // divide by zero is not a semantic error
        }
        // return type check: when error, type is still Int, not Object
    };

    f():Int{
        // neg : incorrect type of argument
        ~"111"
        //  return type check: when error, type is still Int, not Object
    };

    g():Int{
        // lt & leq : incorrect type of argument
        {
            1 < "a";
            true <= "b";
        }
        // return type check: when error, type is still Bool, not Object
    };

    h():Int{
        // comp : incorrect type of argument
        not 123
    };

    i():Int{
        // eq: check all combination
        {
            // 1. both are not Int or Bool or String. should be OK;
```

```
            b = c;
            // 2. one is Int
            1 = b;
            // 3. one is Int, the other is String
            1 = "a";
        }
        // return type: when error, type is still Bool, not Object
    };

    j():Int{
        // loop : check pred type
        while 1 loop b pool
    };

    k():Int{
        // cases/branch:
        {
            // 1. check if there are duplicate branches
            // (i.e. the same type declaration,not the id)
            case 1 of
                x:Int => 1;
                x:Int => 2;
                x:Int => new E;
            esac;

            // 2. check the type of the whole expression
            case 1 of
                x:B => x;
                x:B => 1;
                y:C => y;
            esac;
            // even the branch is duplicate,
            // its branch type should still be gathered
        }
    };

    l():Bool{
        // let:
        {
            // 1. declared type is undefined
            let a:D <- 1 in a;
            // 2. identifier is 'self'
```

```
            let self:D <- 1 in self;
            // 3. init not conform
            // (note return type is String, i.e. type_decl)
            let a:String <- 1 in a;

            // return type check????
            // OK type is still D
            let a:D <- 1 in a;
        }
        + 1
        // type should be expr's type, unless is undefined
    };

    m():Int{
        // dispatch:
        {
            // 1. undeclared id
            a.type_name();
            // 2. method undefined
            (new A).asdasd(g);
            // 3. actual not conform to formal
            (new A).func(1,"abc");
            // 4. acutal num not the same
            (new A).func(1);
        }
    };

    n():Int{
        // static_dispatch:
        {
            // 1. class undefined
            (new A)@U.asd(g);
            // 2. LHS of '@' not conform to RHS
            (new A)@Main.main(g);
            // 3. undeclared id
            a@Object.type_name();
            // 4. method undefined
            (new A)@Z.m();
            // 5. actual not conform to formal
            (new A)@Z.func(g,"abc");
            // 6. actual num not the same
            (new A)@Z.func(1);
```

```
        }
    };
};
```

8.32 测试：mainTest

```
Class Main inherits A{
    // redefinition of main method for inheritance
    main(a:String):Int{
        2
    };
    main(a:String):Int{
        3
    };
};

Class A{
    // redefinition
    main(a:Int):Int{
        1
    };
    main(b:Bool):String{
        "ab"
    };
};

Class B{
    // check if immediately ends
    a():INT{
        1
    };
};
```

8.33 测试：inheritTest

```
// inheritance cycle
Class E inherits A{

};
class A inherits B{

};

class C inherits A{

};

class B inherits C{

};

class D inherits D{

};

class Main{
    main():Int {
        1
    };
};
```

8.34 测试：selfTest

```
Class Main{
    main():Int{
        1
    };
};

Class A{
    x:SELF_TYPE;
    self:Int;
    a():SELF_TYPE{
        new A       // should not conform(notice)
    };

    b():A{
        self        // conform
    };

    c():SELF_TYPE{  // conform
        new SELF_TYPE
    };
};

Class B inherits A{
    self:String;            // self as attr
    d():SELF_TYPE{
        new B       // should not conform
    };

    e():A{
        self
    };

    f():B{
        self
    };

    // we should notice:this is only static analysis
    // so dispatch only considers the return type of the function
    g():B{
```

```
        (new A).c()     // A not conform B
    };

    h():B{
        (new B).c()     //  B conforms B
    };

    i():SELF_TYPE{
        (new A).c()     // A not conform SELF_TYPE
    };

    j():SELF_TYPE{
        (new B).c()     // B not conform SELF_TYPE
    };

    k():B{
        self.c()        // SELF_TYPE conform B
    };

    m():SELF_TYPE{
        self.c()        // should conform(SELF_TYPE conform SELF_TYPE)
        // detail of typing rule: the dispatch type is T_0,not T_0'
        // (i.e. don't convert)
    };

    n():SELF_TYPE{      // SELF_TYPE
        x
    };

    o():Int{
        let a:SELF_TYPE in
            if true then self else self fi
            // LCA of SELF_TYPE and SELF_TYPE is SELF_TYPE
    };

    p():Int{
        case 1 of
            x:Int => self;
            y:String => self;       // same as above
            z:Bool => self;
        esac
    };
```

```
    q(self:Int):Int{        // self as formal:still SELF_TYPE
        self
    };

    r():String{
        let self:Int in self        // self as let id:still SELF_TYPE
    };

    s():Bool{
        case 1 of
            self:Int => self;       // self is Int now???
        esac
    };
};
```

8.35 测试：scopeTest

```
Class Main{
    main():Int{
        1
    };
};

Class A{
    x:Int;
    y:String;
    z:Bool;
    a():Int{
        1
    };
};

Class B inherits A{
    a():Int{
        // 1. should be able to use parent class's attrs and methods
        {
            x <- 1;
            y <- "abc";
            z <- true;
        }
    };

    b():Int{
        // 2. let test
        let x : Int, value : Int <- x + 1 in let value : Int <- 1 in x + value
    };

    c():Int{
        // 3. case test
        case 1 of
            y:Int => y + 1;        // should hide attr y
            a:Bool => 1;
            // b:String => a;      // a should be undeclared
        esac
    };
```

```
    // 4. method formal test
    // should hide attr x and y
    d(x:String,y:Int):Int {
        {
            x <- "abc";
            y <- 1;
        }
    };
};
```

8.36 测试：good

```
Class Main{
    main():Int{
        1
    };
};

Class C inherits B{
    a():SELF_TYPE{
        self.func(y,x)
    };
};

Class B inherits A{
    z:Bool;
    func(a:String,b:Bool):SELF_TYPE{
        self@A.func(a,b)
    };
};

Class A{
    x:Bool;
    y:String;
    func(x:String,y:Bool):SELF_TYPE{
        self
    };
};
```

9. 测试截图

9.1 classTest

```
classTest.cl:12: Redefinition of basic class SELF_TYPE.
classTest.cl:17: Class C was previously defined.
classTest.cl:23: Redefinition of basic class Int.
classTest.cl:27: Redefinition of basic class Int.
classTest.cl:31: Redefinition of basic class String.
classTest.cl:35: Redefinition of basic class Bool.
classTest.cl:39: Redefinition of basic class IO.
classTest.cl:43: Redefinition of basic class Object.
classTest.cl:47: Redefinition of basic class SELF_TYPE.
classTest.cl:7: Class C cannot inherit class SELF_TYPE.
classTest.cl:2: Class A inherits from an undefined class B.
Compilation halted due to static semantic errors.
```

9.2 methodTest

```
methodTest.cl:16: In redefined method func, return type String is different from original return type Int.
methodTest.cl:24: In redefined method func, return type String is different from original return type Int.
methodTest.cl:31: Incompatible number of formal parameters in redefined method func.
methodTest.cl:38: In redefined method func, parameter type Bool is different from original type Int
methodTest.cl:44: Incompatible number of formal parameters in redefined method func.
methodTest.cl:47: Incompatible number of formal parameters in redefined method func.
methodTest.cl:84: Method e is multiply defined.
methodTest.cl:56: 'self' cannot be the name of a formal parameter.
methodTest.cl:56: 'self' cannot be the name of a formal parameter.
methodTest.cl:56: Inferred return type SELF_TYPE of method a does not conform to declared return type Int.
methodTest.cl:60: Formal parameter self cannot have type SELF_TYPE.
methodTest.cl:60: 'self' cannot be the name of a formal parameter.
methodTest.cl:64: Formal parameter b is multiply defined.
methodTest.cl:69: Class INT of formal parameter x is undefined.
methodTest.cl:69: Class INT of formal parameter x is undefined.
methodTest.cl:69: Formal parameter x is multiply defined.
methodTest.cl:69: Class STR of formal parameter self is undefined.
methodTest.cl:69: 'self' cannot be the name of a formal parameter.
methodTest.cl:74: 'self' cannot be the name of a formal parameter.
methodTest.cl:74: Class INT of formal parameter self is undefined.
methodTest.cl:74: 'self' cannot be the name of a formal parameter.
methodTest.cl:74: Undefined return type INT in method d.
methodTest.cl:79: 'self' cannot be the name of a formal parameter.
methodTest.cl:79: Class INT of formal parameter self is undefined.
methodTest.cl:79: 'self' cannot be the name of a formal parameter.
methodTest.cl:79: Inferred return type String of method e does not conform to declared return type Int.
Compilation halted due to static semantic errors.
```

9.3 attrTest

```
attrTest.cl:13: Attribute b is multiply defined in class.
attrTest.cl:19: Attribute a is an attribute of an inherited class.
attrTest.cl:20: Attribute a is an attribute of an inherited class.
attrTest.cl:9: Class INT of attribute a is undefined.
attrTest.cl:9: Undeclared identifier c.
attrTest.cl:11: Inferred type String of initialization of attribute b does not conform to declared type Int.
Compilation halted due to static semantic errors.
```

9.4 exprTest

```
exprTest.cl:26: Assignment to undeclared variable a.
exprTest.cl:26: Undeclared identifier a.
exprTest.cl:30: Type String of assigned expression does not conform to declared type B of identifier b.
exprTest.cl:33: Inferred return type C of method c does not conform to declared return type Int.
exprTest.cl:42: Predicate of 'if' does not have type Bool.
exprTest.cl:39: Inferred return type Main of method d does not conform to declared return type Int.
exprTest.cl:49: non-Int arguments: Int + String
exprTest.cl:50: non-Int arguments: String - Int
exprTest.cl:51: non-Int arguments: String * Bool
exprTest.cl:52: non-Int arguments: String / Int
exprTest.cl:59: Argument of '~' has type String instead of Int.
exprTest.cl:66: non-Int arguments: Int < String
exprTest.cl:67: non-Int arguments: Bool <= String
exprTest.cl:63: Inferred return type Bool of method g does not conform to declared return type Int.
exprTest.cl:74: Argument of 'not' has type Int instead of Bool.
exprTest.cl:72: Inferred return type Bool of method h does not conform to declared return type Int.
exprTest.cl:83: Illegal comparison with a basic type.
exprTest.cl:85: Illegal comparison with a basic type.
exprTest.cl:77: Inferred return type Bool of method i does not conform to declared return type Int.
exprTest.cl:92: Loop condition does not have type Bool.
exprTest.cl:90: Inferred return type Object of method j does not conform to declared return type Int.
exprTest.cl:101: Duplicate branch Int in case statement.
exprTest.cl:102: Duplicate branch Int in case statement.
exprTest.cl:102: 'new' used with undefined class E.
exprTest.cl:108: Duplicate branch B in case statement.
exprTest.cl:95: Inferred return type Object of method k does not conform to declared return type Int.
exprTest.cl:119: Class D of let-bound identifier a is undefined.
exprTest.cl:121: Class D of let-bound identifier self is undefined.
exprTest.cl:121: 'self' cannot be bound in a 'let' expression.
exprTest.cl:123: Inferred type Int of initialization of a does not conform to identifier's declared type String.
exprTest.cl:127: Class D of let-bound identifier a is undefined.
exprTest.cl:129: non-Int arguments: D + Int
exprTest.cl:115: Inferred return type Int of method l does not conform to declared return type Bool.
exprTest.cl:137: Undeclared identifier a.
exprTest.cl:139: Undeclared identifier g.
exprTest.cl:139: Dispatch to undefined method asdasd.
exprTest.cl:141: In call of method func, type String of parameter b does not conform to declared type Int.
exprTest.cl:143: Method func called with wrong number of arguments.
exprTest.cl:151: Undeclared identifier g.
exprTest.cl:151: Static dispatch to undefined class U.
exprTest.cl:153: Undeclared identifier g.
exprTest.cl:153: Expression type A does not conform to declared static dispatch type Main.
exprTest.cl:155: Undeclared identifier a.
exprTest.cl:157: Static dispatch to undefined method m.
exprTest.cl:159: Undeclared identifier g.
exprTest.cl:159: In call of method func, type Object of parameter a does not conform to declared type Int.
exprTest.cl:159: In call of method func, type String of parameter b does not conform to declared type Int.
exprTest.cl:161: Method func invoked with wrong number of arguments.
Compilation halted due to static semantic errors.
```

9.5 mainTest

```
mainTest.cl:15: Method main is multiply defined.
mainTest.cl:2: In redefined method main, parameter type String is different from original type Int
mainTest.cl:5: In redefined method main, parameter type String is different from original type Int
mainTest.cl:1: No 'main' method in class Main.
mainTest.cl:21: Undefined return type INT in method a.
Compilation halted due to static semantic errors.
```

9.6 inheritTest

```
inheritCycle.cl:17: Class D, or an ancestor of D, is involved in an inheritance cycle.
inheritCycle.cl:13: Class B, or an ancestor of B, is involved in an inheritance cycle.
inheritCycle.cl:9: Class C, or an ancestor of C, is involved in an inheritance cycle.
inheritCycle.cl:5: Class A, or an ancestor of A, is involved in an inheritance cycle.
inheritCycle.cl:2: Class E, or an ancestor of E, is involved in an inheritance cycle.
Compilation halted due to static semantic errors.
```

```
inheritCycle.cl:17: Class D, or an ancestor of D, is involved in an inheritance cycle.
inheritCycle.cl:9: Class C, or an ancestor of C, is involved in an inheritance cycle.
inheritCycle.cl:13: Class B, or an ancestor of B, is involved in an inheritance cycle.
inheritCycle.cl:5: Class A, or an ancestor of A, is involved in an inheritance cycle.
inheritCycle.cl:2: Class E, or an ancestor of E, is involved in an inheritance cycle.
Compilation halted due to static semantic errors.
```

9.7 selfTest

```
selfTest.cl:9: 'self' cannot be the name of an attribute.
selfTest.cl:24: 'self' cannot be the name of an attribute.
selfTest.cl:10: Inferred return type A of method a does not conform to declared return type SELF_TYPE.
selfTest.cl:25: Inferred return type B of method d does not conform to declared return type SELF_TYPE.
selfTest.cl:39: Inferred return type A of method g does not conform to declared return type B.
selfTest.cl:47: Inferred return type A of method i does not conform to declared return type SELF_TYPE.
selfTest.cl:51: Inferred return type B of method j does not conform to declared return type SELF_TYPE.
selfTest.cl:68: Inferred return type SELF_TYPE of method o does not conform to declared return type Int.
selfTest.cl:73: Inferred return type SELF_TYPE of method p does not conform to declared return type Int.
selfTest.cl:81: 'self' cannot be the name of a formal parameter.
selfTest.cl:81: Inferred return type SELF_TYPE of method q does not conform to declared return type Int.
selfTest.cl:86: 'self' cannot be bound in a 'let' expression.
selfTest.cl:85: Inferred return type SELF_TYPE of method r does not conform to declared return type String.
selfTest.cl:91: 'self' bound in 'case'.
selfTest.cl:89: Inferred return type Int of method s does not conform to declared return type Bool.
Compilation halted due to static semantic errors.
```

9.8 scopeTest

```
_block
  #20
  _assign
    x
    #20
    _int
      1
    : Int
  : Int
  #21
  _assign
    y
    #21
    _string
      "abc"
    : String
  : String
  #22
  _assign
    z
    #22
    _bool
      1
```

```
x
Int
#0
_no_expr
: _no_type
#30
_let
  value
  Int
  #30
  _plus
    #30
    _object
      x
    : Int
    #30
    _int
      1
    : Int
  : Int
  #30
  _let
    value
    Int
    #30
    _int
      1
    : Int
    #30
    _plus
      #30
      _object
        x
      : Int
      #30
      _object
        value
      : Int
    : Int
  : Int
: Int
Int
```

9.9 good

```
    _method
     func
     #15
     _formal
       a
       String
     #15
     _formal
       b
       Bool
     SELF_TYPE
     #16
     _static_dispatch
       #16
       _object
         self
       : SELF_TYPE
       A
       func
       (
       #16
       _object
         a
       : String
       #16
       _object
         b
       : Bool
       )
     : SELF_TYPE
   )
 #20
 _class
   A
   Object
   "good.cl"
   (
   #21
   _attr
     x
     Bool
     #0
     _no_expr
     : _no_type
   #22
   _attr
     y
     String
```