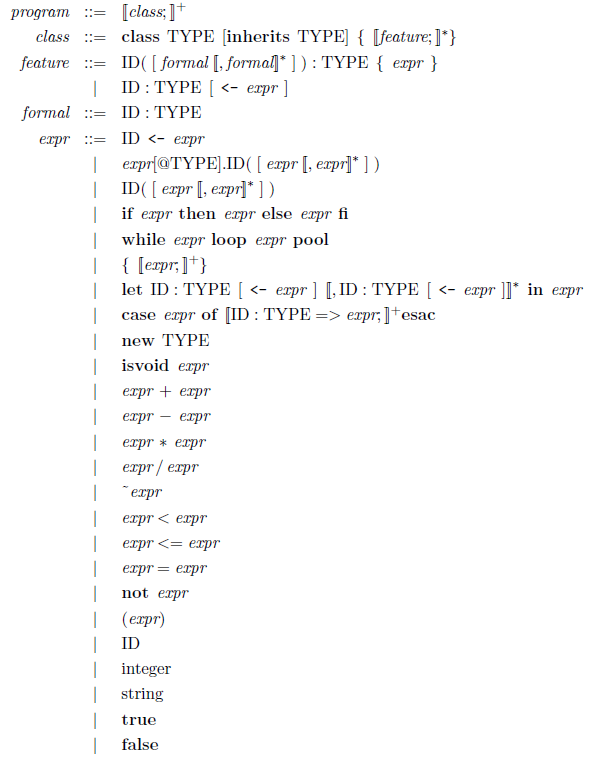
**编译PA2：词法分析 报告**

17307130015 王鑫涛

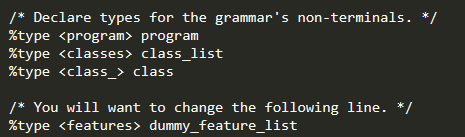
**一、背景知识**

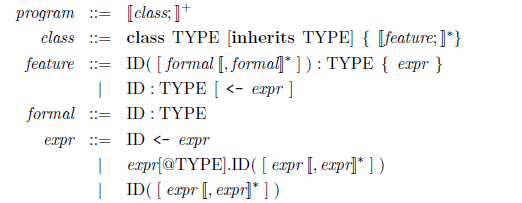
PA2的目的是输出一棵AST。主要内容可以参考cool-manual中的第12部分，我们需要完成这些类型规则。



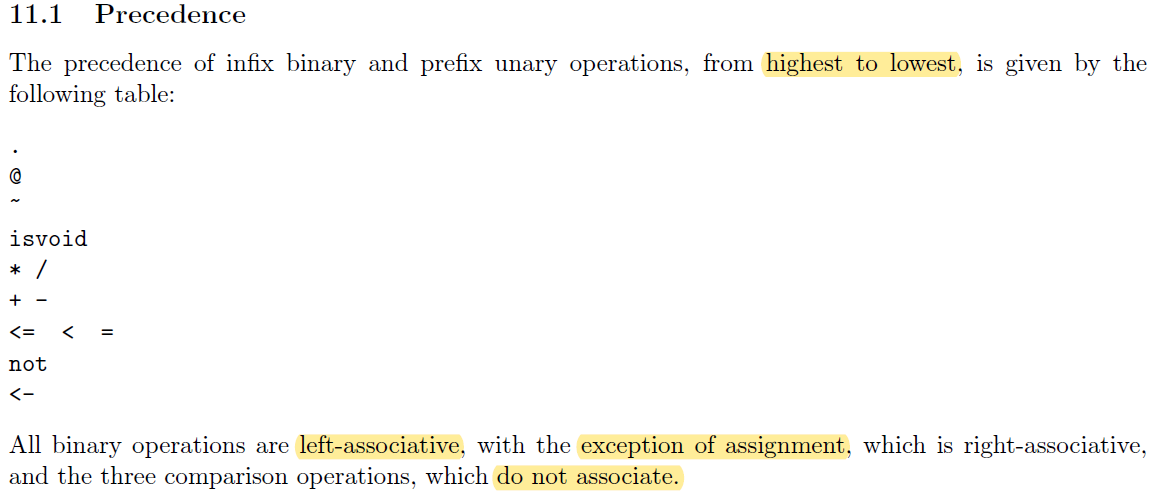
**二、非终结符类型声明以及优先级定义**

声明区首先要求我们为语法的非终结符声明类型。由于You will want to change the following line.和上面的declare注释缩进一样，具备一定的迷惑性。其实you will下面是我们需要定义的非终结符类型，而上面则是已经定义好的。根据manual-cool.pdf中的第12部分，可知我们需要定义的类型为program、class、feature、formal和expression五类。其中，根据cool-manual.pdf中的3.1，feature的含义是一个类的一个属性或一个方法。而formal则可以推断是是方法中的一个参数，包含参数的名字和参数的类型。

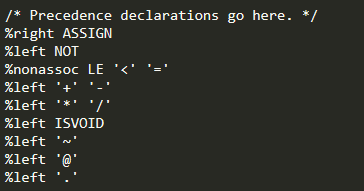




根据cool-manual.pdf中的11.1，为操作符定义优先级。



在bison中，优先级是先定义的优先级低，后定义的优先级高。根据上述文档，而元运算符中仅有赋值运算符是右结合的，其余均为左结合。三个比较运算符则不具有结合律。一元运算符NOT也是右结合的。



**三、语法规则：程序和类**

在语法规则中，需要介绍一些符号。

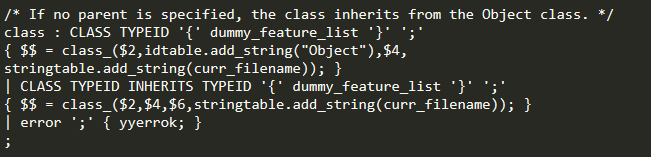


以这句话为例。在为非终结符class定义时，有产生式CLASS TYPEID ‘{‘ dummy\_feature\_list ‘}’ ‘;’，后面的花括号内是其对应的行为。这里用到的参数$2和$4，则分别表示产生式右边中第二个和第四个元素的值。

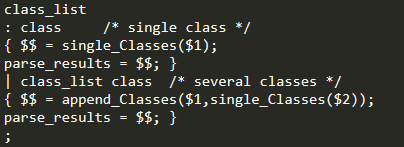
接下来，依次介绍程序和类的定义，以进一步了解语法规则。



程序的产生式即class\_list，即一个程序即若干个类定义。@$表示产生式左边的符号的行号，@1则表示产生式右边第一个元素的行号。对于非终结符，我们可以选择产生式中任意元素的行号，而如果我们省略它，bison也会自动为我们决定。匹配到program的行为即用第一个参数的值（class\_list）去初始化一个program，作为ast的根。

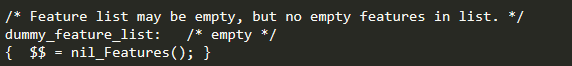


类别的定义有2种。第一种是非继承的，第二种是继承的。后者在产生式种多了2个元素INHERITS TYPEID，其中TYPEID（即$4）被作为初始化类class\_的参数。同时，这里还需要进行错误情况的匹配，并以yyerrok要求在检测到错误之后还会继续匹配。



class\_list即可以是一个单一的class，也可以通过递归定义class\_list : class\_list class，表示为多个class。$$为产生式左边的值。

**四、语法规则：特征Feature**

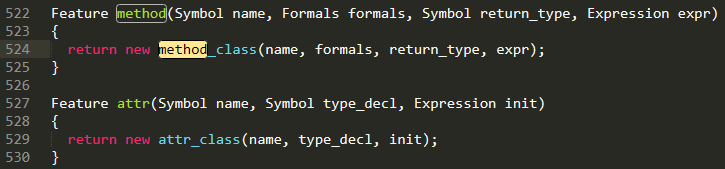


首先看提供给我们的样例dummy\_feature\_list。考虑到dummy\_feature\_list在class定义中的出现，可知dummy的意思是还未定义好而需要我们完善的。（注释有点拗口）可以看出，它首先为我们定义好了feature list为空的情况，即什么都没匹配到的话，会调用nil\_Features， 这个函数在cool-tree.cc中定义，会new一个nil节点。

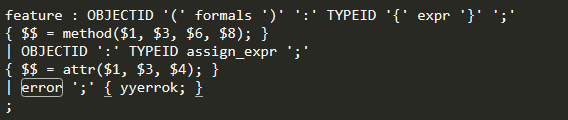


根据cool-manual.pdf中给出的产生式，可知一个feature应包含ID:TYPE及可选的表达式（方括号表示可选，而非括号字符本身），或者ID，圆括号包起来的若干个formal（虽然其定义看起来是1个+0或多个，但其实因为用了方括号表示可选，也可以是0个），冒号，TYPE，以及花括号包起来的表达式。前者是一个方法，其formal是参数，TYPE是返回值的类型，花括号里的expr是方法的行为。后者是属性，可选的expr用以初始化。

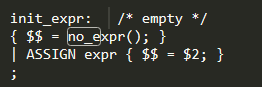
因为class的定义中，调用了class\_这个函数，我想定义feature时也应该有相应的函数。于是我先在cool-tree.cc中找到了class\_的构造函数，然后也在这个文件中找到了返回Feature的两个函数：method和attr。



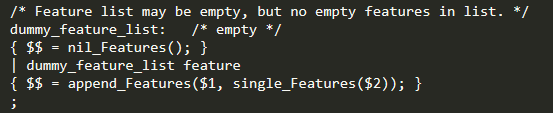
按照cool-manual中给出的产生式定义feature，并调用上面2个方法。可以看到，在调用方法时，我们其实只需要把有意义的参数依次传入就行了。



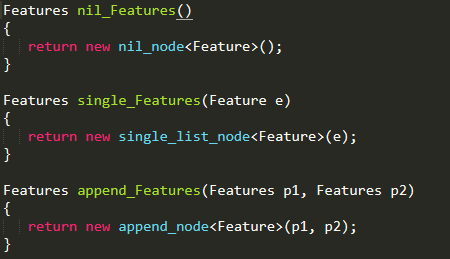
这里的init\_expr，由于是可选的，所以也可以是空的。在空的情况下，参考dummy\_feature\_list，什么都不匹配，相应地调用cool-tree.cc中的no\_expr函数。非空的情况下，将匹配ASSIGN expr，只需要将expr的值赋予init\_expr即可。formal\_list将在定义完formal后定义。



定义好feature之后，就可以完善dummy\_feature\_list的定义。我们定义好了空的情况，接下来要定义非空情况，只需要递归地定义dummy\_feature\_list: dummy\_feature\_list feature，即可表示有若干个feature的情况。

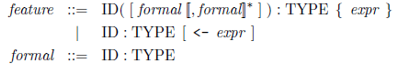


这里调用的函数为cool-tree.cc中的append\_Features。由于$1是nil\_Features产生的node，类型为Features，而feature本身类型为Feature（单数），还需要通过使用single\_Features包装为Features，才能传入append\_Features。

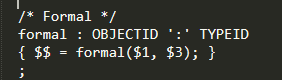


**五、语法规则：Formal**

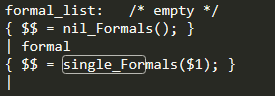
formal是方法中的一个参数，包含参数的名字和参数的类型。其产生式和使用到其的产生式主要是这2条。



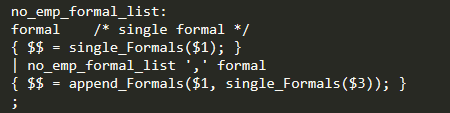
formal的定义比较简单。



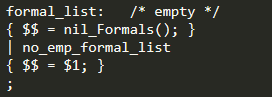
接下来需要实现feature中用到的formal\_list。首先，最简单的情况是空，或者单一的formal。这一步比较简单。但第三种情况是，formal, formal, formal, ….。这里需要处理这个逗号。而如果直接用formal\_list: formal\_list ‘,’ formal，则会匹配到 , formal，这是我们所不期望的。



所以定义一个辅助的非终结符no\_emp\_formal\_list，以规避掉空情况，就不会匹配到, formal。这里的处理和dummy\_feature\_list基本一致。



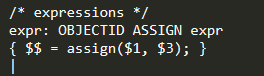
再进行完这一步后，我们发现，由于no\_emp\_formal\_list包含了一条formal和多条formal的情况，刚才在formal\_list中定义的一条formal的情况就是多余的了，故去掉。



**六、语法规则：Expression**

Expression的产生式是最为复杂的，足足有二十多行。这一部分可以参考cool-manual中的7。

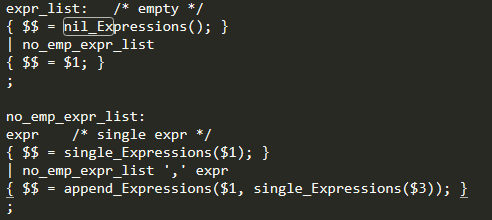
expr产生式的第一行是赋值，非常简单，调用cool-tree.cc中的assign函数即可。



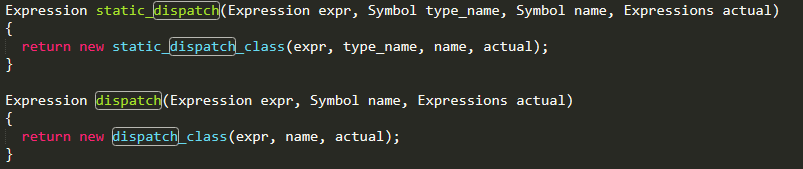
注意到expr产生式的第二行、第三行出现了[expr [[, expr]] \*]，这表示0个expr、1个expr或多个以逗号分隔的expr，这和上面定义的formal\_list非常像。



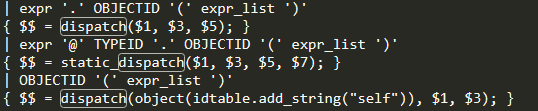
相应的，我们就可以定义expr\_list和no\_emp\_expr\_list。这部分的思想和formal\_list基本一致，不再赘述。



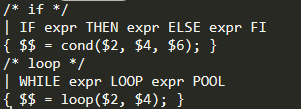
第二行和第三行的含义是dispatch（参见cool-manual的7.4），意即方法调用。如果忽略第二行的@TYPE，即调用expr下id这个object（方法）。第二行中可选的@TYPE提供了一个指定选择调用”哪个父类的该方法”的途径，即指定使用类型TYPE的该方法，而非使用运行时的动态类型。第三行没有expr，而是直接使用ID，实际上是调用了self.id的方法，省略了self。



相应的函数如上图所示。调用这些函数即可。



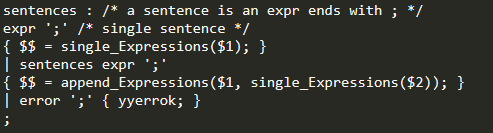
接下来是if语句和while语句，也较为简单。对应的函数是cond和loop。



然后第六条是块定义。包含一对花括号包起来的至少1个expr ;



先使用左递归的方式定义花括号内的内容。因为使用了分号，我认为一个分号将一个表达式变为一条语句，故称其为一条sentence，所以定义非终结符sentences。相应的函数为block。

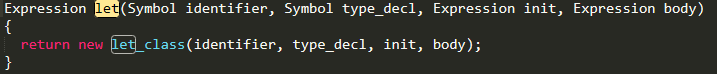


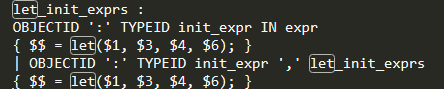


第七条语句是Let语句。每一个<idk> :<typek> [<-<exprk> ]为IDK这个变量指定类型，并（可选地）用一个表达式进行初始化。Let语句将从左向右依次计算各个表达式，最后计算expr。



由于let语句调用的let函数，需要指定body，即最后一个expr，即右边的IN expr是固定的，所以这部分中间语句不能使用左递归实现，只能先将左边的let拆开，使用右递归。

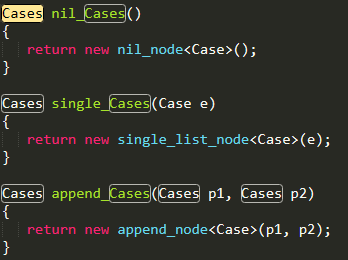


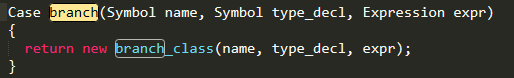


第八条是case语句，至少有一个case。

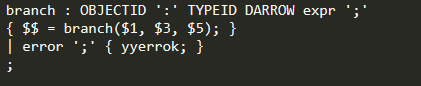


为了实现它，首先对cool-tree.cc中相关的函数要有一定的了解。

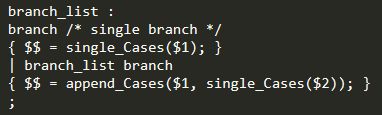




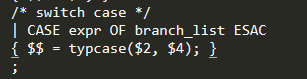
首先为branch定义产生式，并调用branch这个函数。需要传入的参数有OBJECTID TYPEID和expr。

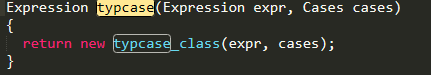


随后按左递归的方法构造branch\_list，用以存储多个branch。这里需要将branch构造为Cases。参考classes。

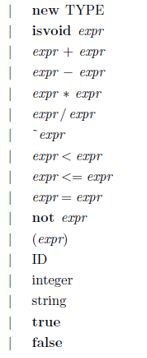
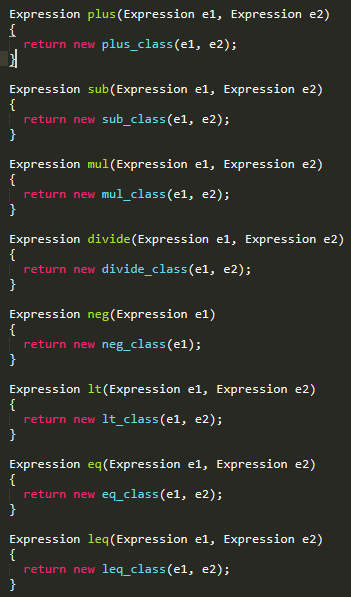
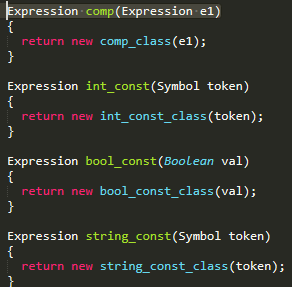


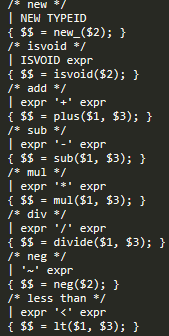
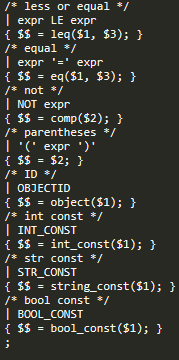
定义好了branch\_list之后，就可以定义expr中case语句的产生式。调用typcase函数，typcase函数接收表达式expr和branch\_list。





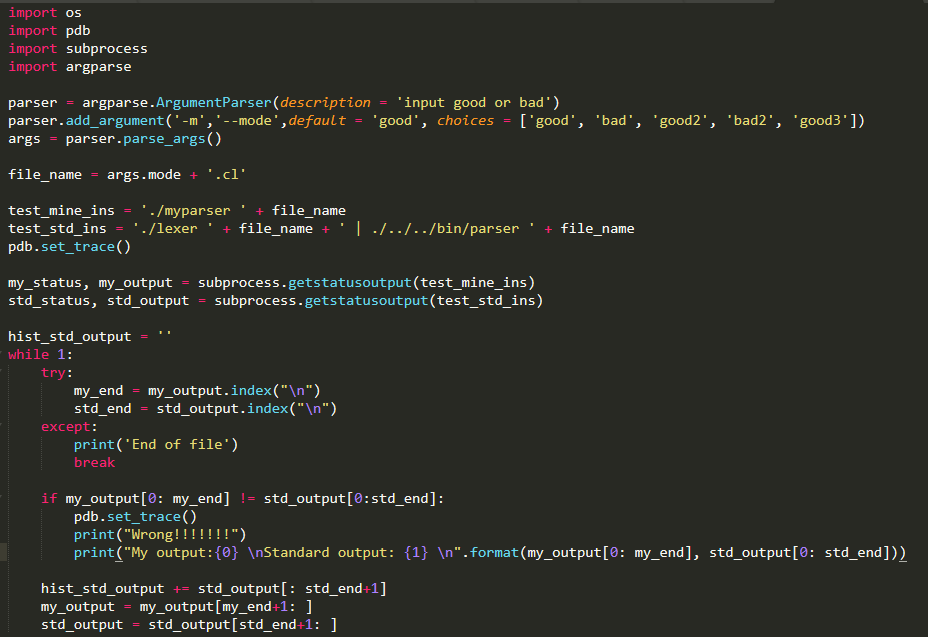
之后的产生式都较为trivial。需要调用的函数在cool-tree.cc中几乎是依次定义好了。

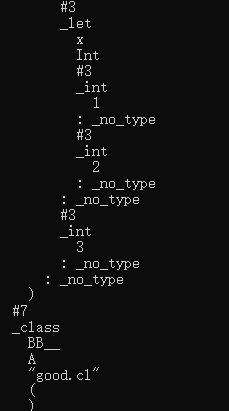
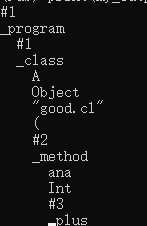
 

**七、实验**

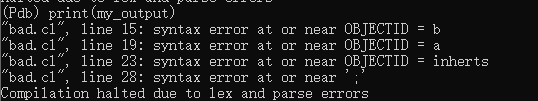
和PA1一样，为了进行实验，写了一个用于比较我的parser输出和标准parser输出的python代码。可以用-m指定需要测试的文件。使用hist\_std\_output来存储历史输出信息，以便debug。



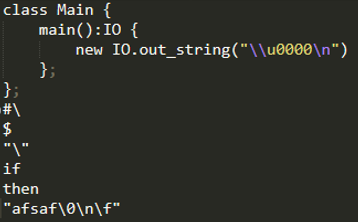
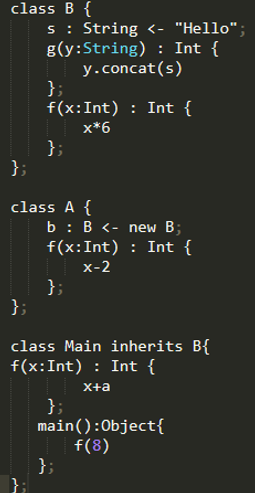
给定good.cl的输出如下。



给定bad.cl的输出如下。



根据cool语法，修改good.cl和bad.cl，得到good2.cl和bad2.cl（分别在左、右）。实验结果也是正确的。



**八、心得体会**

在PA2中，我们需要处理的代码更多了，需要对产生式有更多的理解。一开始上手时，确实觉得非常头疼，尤其是有很多不明所以的函数。如nil\_Features()。在实验中，逐渐发现，本次使用到的大部分函数都在cool-tree.cc中，并且具有较为一致的实现。经过对cool-tree.cc和cool-manual.pdf不断阅读，会发现自己对cool的理解越来越多。在处理很多表达式的时候，会发现对表达式的不理解、对调用什么函数不清楚，这个时候去看cool-manual.pdf往往能起到帮助，而cool-tree.cc中也包含许多重要的信息。

在实验中，对错误的处理是比较难理解的一个部分。因为我们不止要生成正确的结果，对错误的测试样例，我们也希望生成的报错信息能和标准parser一致。因此，就需要使用到对错误的yyerrok，使模型在检测到错误时能不停止，这个问题花了我比较多的时间。