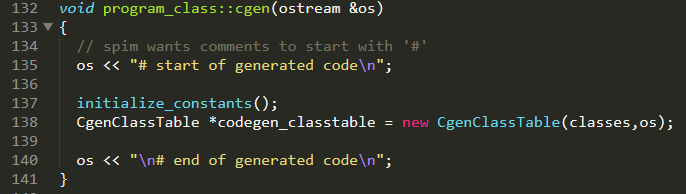
**编译PA4：代码生成 报告**

17307130015 王鑫涛

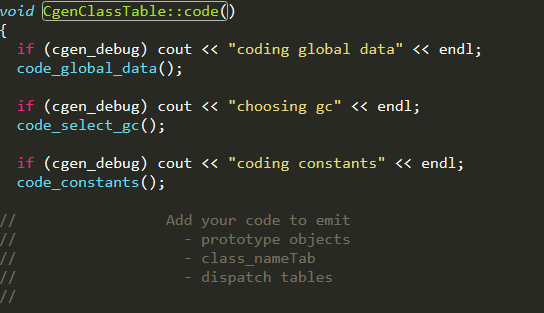
**一、背景知识**

PA4的任务是代码生成。

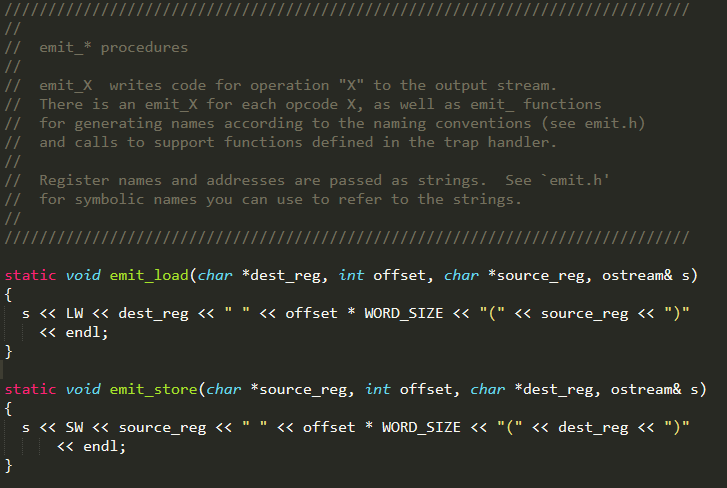
我们要处理的函数的入口为program\_class:cgen，应在这里调用CgenClassTable::code()函数。



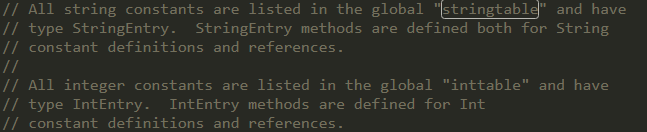
code函数逐步为程序各部分内容生成代码。code\_global\_data()、code\_select\_gc()和code\_constants()是已经为我们定义好了的，可以作为参考。



虽然说，PA4需要我们生成MIPS汇编语言的代码，但实际上，我们不需要具体地搞清楚MIPS中每条指令是怎么写的，因为emit\_\*函数帮我们定义好了指令是怎么写的。当我们想要写一条指令X时，只需要用相应的参数调用emit\_X即可。



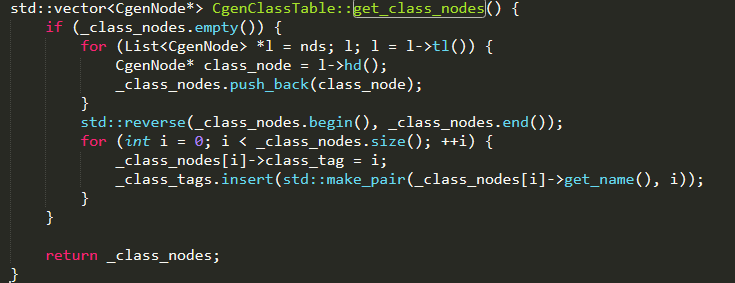
所有的字符串常量都被记录在全局变量stringtable中，对应一个StringEntry的对象。同理，所有整形常量都被记录在全局变量inittable中，对应一个IntEntry。



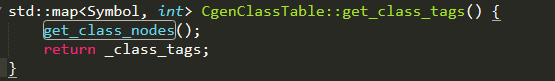
代码整体可以分为两部分，一部分是构建CgenClassTable、CgenNode等数据结构，一部分是服务于代码的生成。

**二、CgenClassTable和CgenNode上的数据访问**

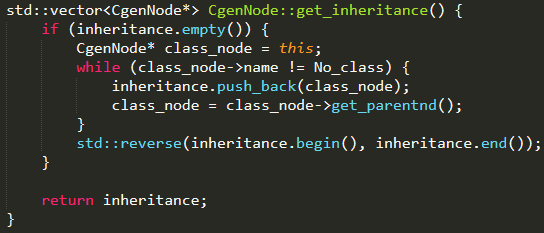
函数get\_class\_nodes，该函数第一次调用会将CgenClassTable的链表lds用以构建一个CgenNode的向量和一个从Symbol类名到int（其下标）的map。通过get\_class\_nodes ()函数，可以得到所有类别的节点。



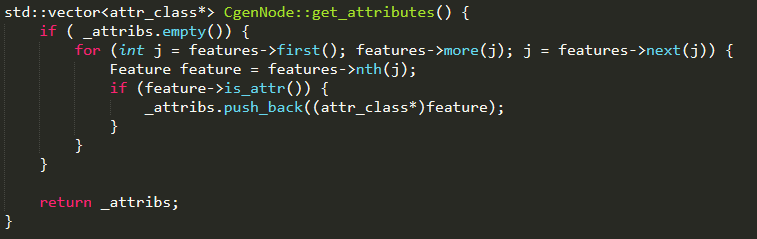
get\_class\_tags函数希望得到get\_class\_nodes中生成的那个从Symbol类名到int（其下标）的map，为了避免该map尚未被创建，所以需要先调用一次get\_class\_nodes，如果该map已被创建，get\_class\_nodes将什么也不会干。



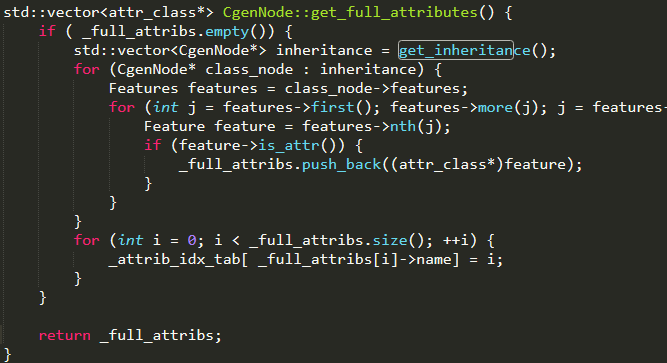
节点上定义有函数get\_inheritance，用以获取节点的继承关系。当第一次调用时，会自底向上从父类开始遍历class\_node的祖先类，将其加入一个向量inheritance当中，随后reverse，获取从第一个祖先开始，到该节点的父类的向量。



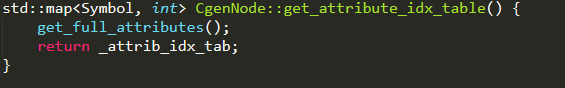
节点上的函数get\_attributes用以遍历节点所有feature，将其中的attr加入到\_attribs这个向量中。



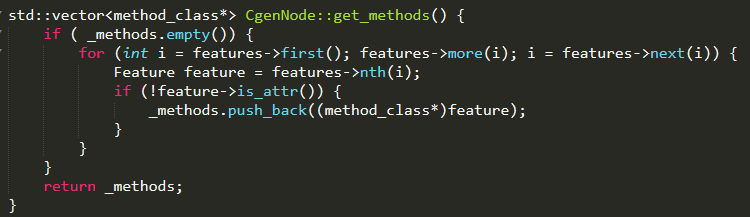
get\_full\_attributes函数则会将get\_inheritance里得到的所有父类的attr，加入到\_full\_attribs向量中。同时，创建一个从attr的名字到其下标的map。



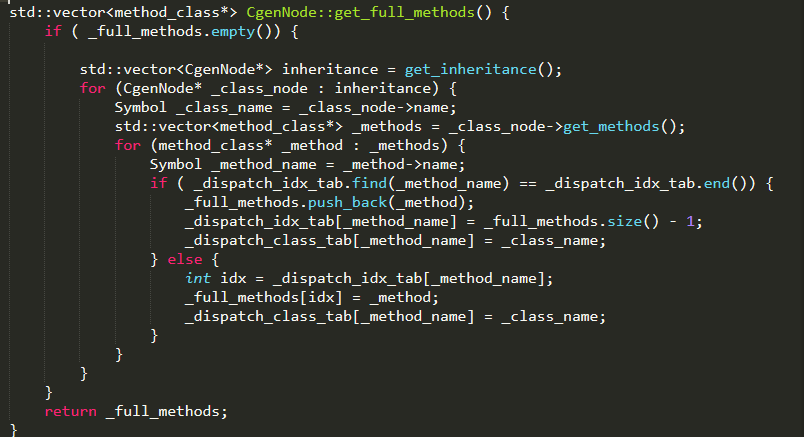
当想要访问上述map时，就先调用一次get\_full\_attributes，如果map已经存在，该函数会直接返回。然后再返回\_attrib\_idx\_tab。



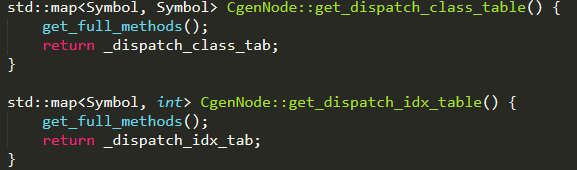
同理，get\_methods函数用以获得类节点下定义的方法。



而get\_full\_methods函数则会将get\_inheritance里得到的所有父类的method，加入到\_full\_methods向量中，并构建method到对应类的map和method到其在向量中所在下标的列表。如果父类和子类中重复定义了该method时，由于是自顶向下访问inheritance数组，后来的（子类的）方法会覆盖父类的方法。

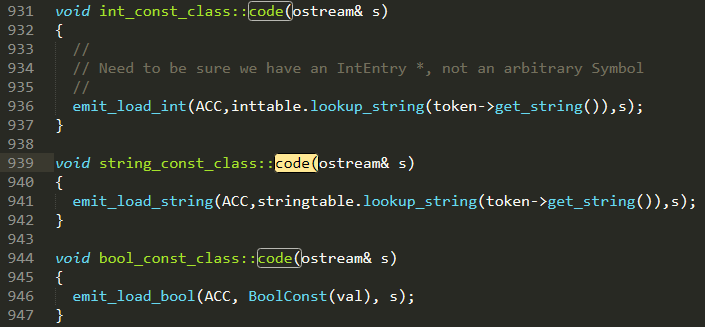


用get\_dispatch\_class\_table和get\_dispatch\_idx\_table访问上述2个map。

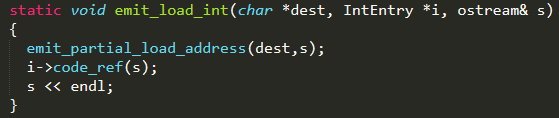


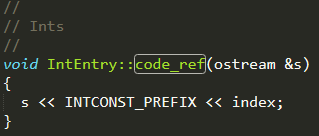
**三、表达式的码生成**

我们需要为各种表达式定义其代码生成的方式。其中，最简单的整形常量、字符串常量及布尔常量表达式的代码生成已经帮我们定义好了，即调用emit\_load\_int、emit\_load\_string和emit\_load\_bool即可。

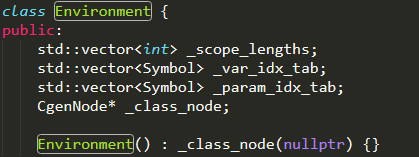


以emit\_load\_int为例，它会调用emit\_partial\_load\_address函数，然后调用IntEntry的code\_ref函数，生成访问这个整形常量的汇编代码。

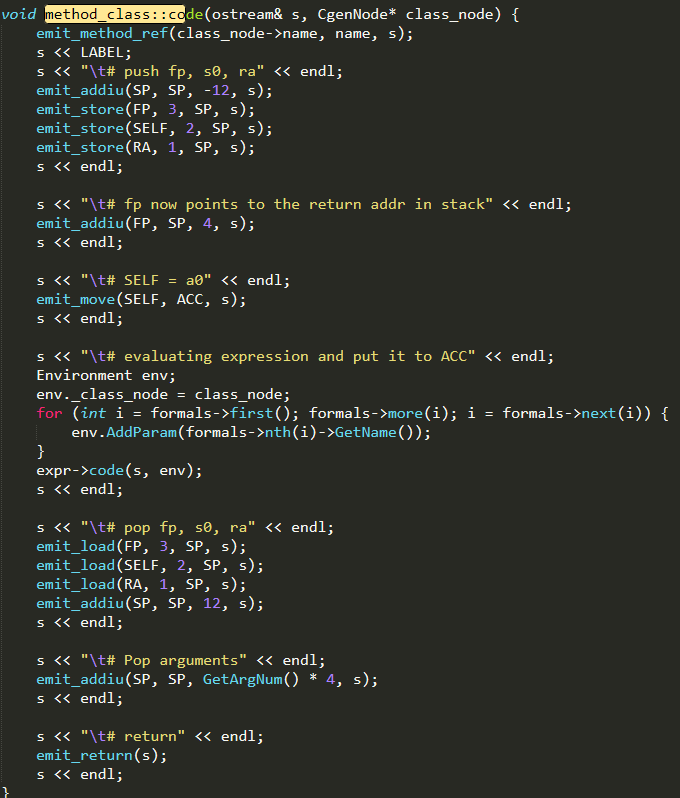




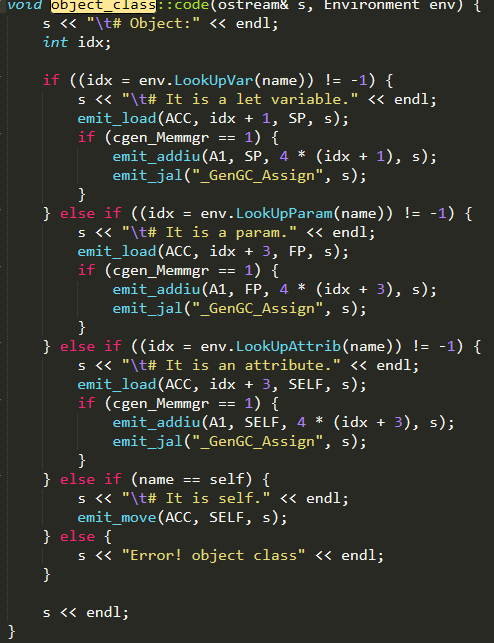
当我们要去为一个方法进行代码生成时，首先会标记一个LABEL，然后构建其上下文。我们把fp, s0, ra作为被调用者保存的寄存器。为了描述一个作用域内的环境，定义了一个类Environment，其属性包含一个类节点\_class\_node、方法的参数的向量等。



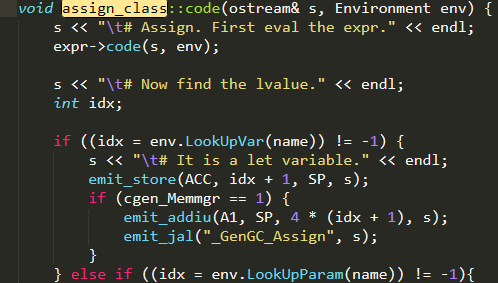
method\_class::code用以为一个方法进行代码生成。函数的主体是编码其中的表达式expr。首先保存被调用者保存的寄存器，创建栈帧，将各formal加入到环境中，然后为expr进行代码生成，再进行寄存器和栈帧的恢复，然后弹出方法的参数，最后return。



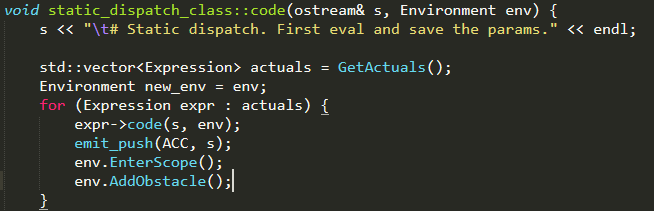
object\_class::code为Object Id进行代码生成，首先要找到这个ID是在哪里定义的。可能的出处是let、formal、attribute和self。我们分别在环境中检查这些地方。找到对应的位置后，从该位置进行load即可。如果开启了GC，则需要完成一次参数防止和函数调用。



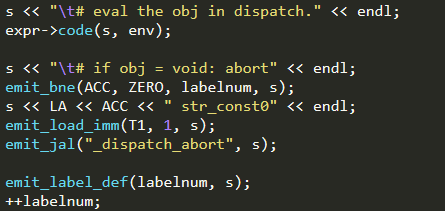
assign\_class::code本质上是一个store语句，将其expr部分先代码生成，按上述方法依次从let变量、参数、属性中寻找其左值的地址，然后向对应地址执行store。



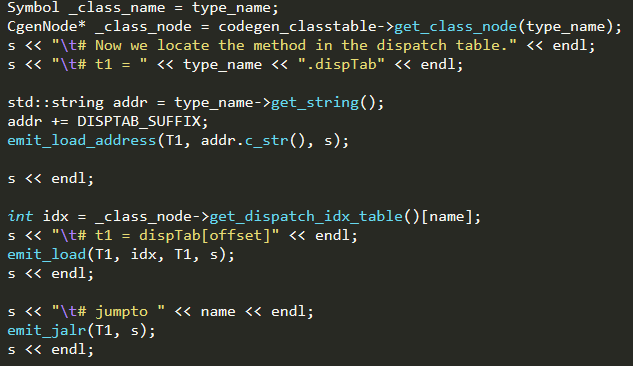
static\_dispatch是一次静态的函数调用，我们为其每个表达式生成代码，将结果按次序存储到栈中。



然后计算其调用的对象，并定义一个LABEL。如果算出来的结果不对应已知的任何类，会将其赋值为0。如果不是0，则bne后jump到LABEL后正常逻辑的执行，否则顺序执行触发ABORT。



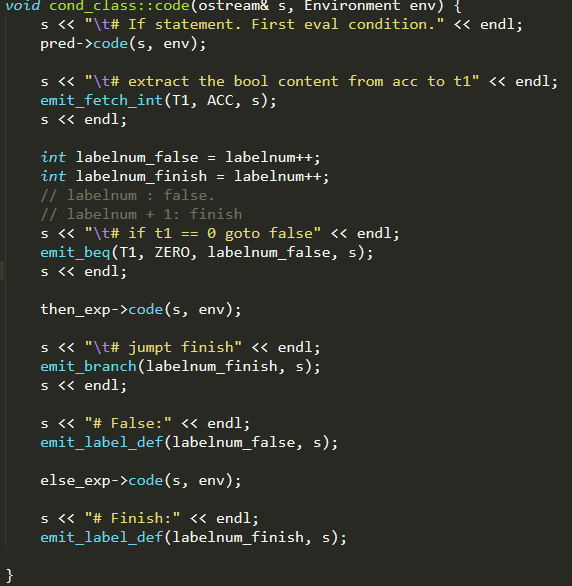
通过get\_class\_node，我们可以找到对象的类别对应的节点，然后定位到该方法。我们在该类别对应的dispatch表中，找到该dispatch的下标，以获得需要调用的函数的地址。然后可以生成一条jalr的指令，跳转到该地址。



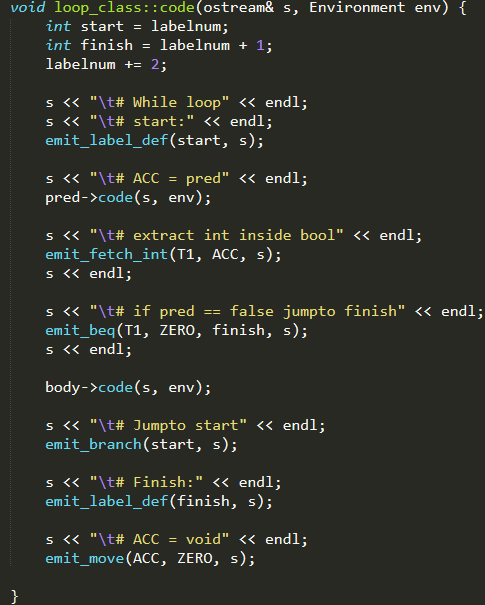
而dispatch\_class::code()与static的差别仅在于，其\_class\_name为表达式expr的类别，而不是静态指定的类别。



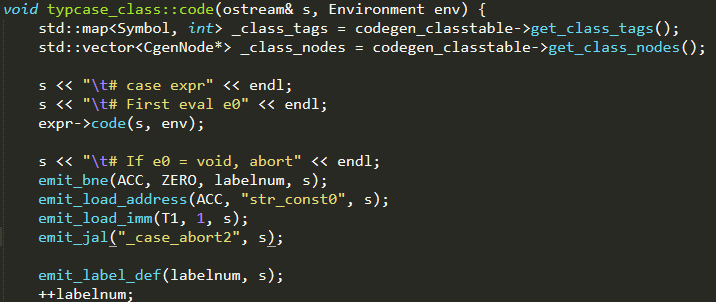
cond\_class::code()要生成条件语句。首先，为计算条件的值生成代码。根据上计算机体系结构时的经验，我选择使用一条beq的语句，使用false和finish两个label。当条件满足时，顺序执行（then后的语句then\_exp），然后jump到finish；当条件不满足时，jump到false（else后的语句else\_exp），然后执行到finish。



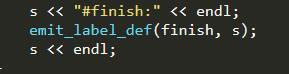
循环语句的代码生成loop\_class::code()和条件语句比较类似。第一个label为start，第二个label为finish。start后为条件语句pred，为其生成代码。当条件成立时，进行循环；当条件不成立时，跳转到finish，退出循环。每当运行完循环体，就会跳转到start，重新进行判断。



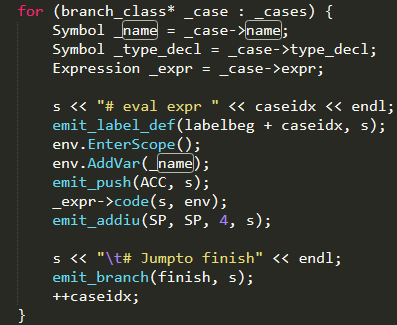
typcase\_class::node()比较复杂。首先，我们获得所有类的节点，然后计算表达式expr。如果表达式的类型为void，就跳转到abort。



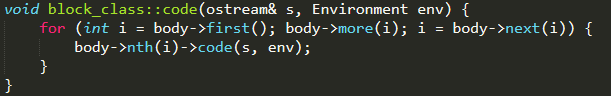
将finish的标签定义在最后。



为每个标签生成代码，在执行完该标签的代码后，jump到finish。



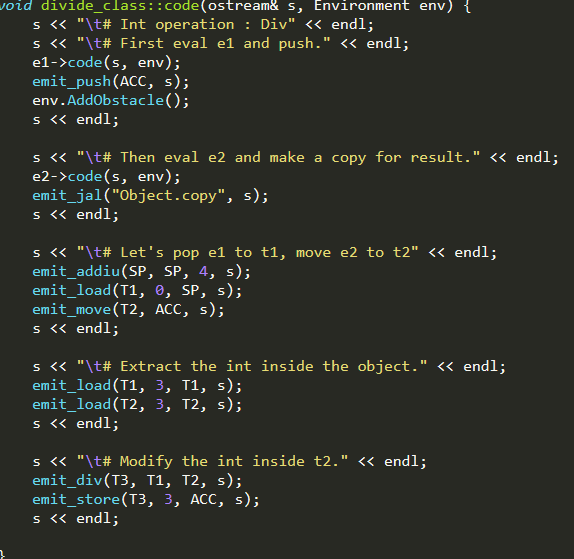
block\_class::code()比较简单，对block内的语句依次进行代码生成即可。



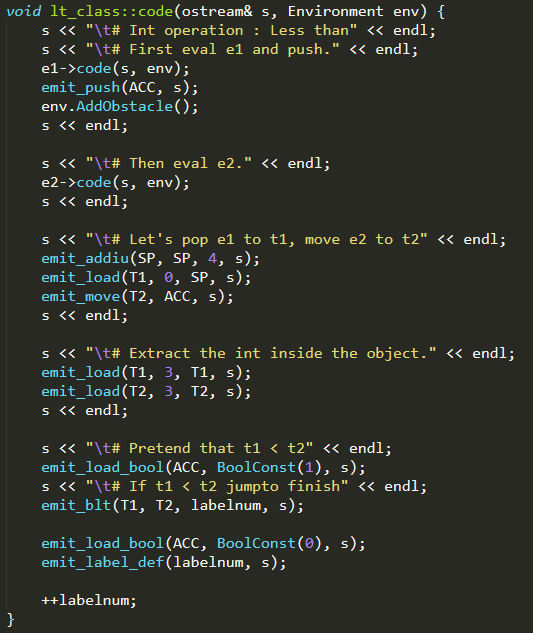
let语句先为初始化表达式生成代码。如果初始化表达式为空，且声明类型为基本类型str、int或bool，则调用对应的load函数。之后先push将let变量加入栈中，再生成函数体的代码，然后再进行pop。



加减乘除运算的code()较为一致。首先计算表达式e1和e2，分别push入栈中。然后将e1和e2分别pop到t1和t2，执行相应地计算并将值存储到t2。

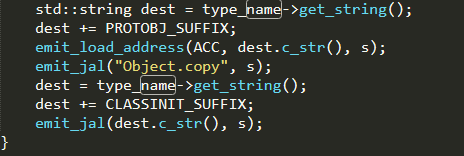


lt、eq、leq三个比较和四则运算也类似，分别计算e1和e2，然后读入t1和t2，进行比较，将结果存入ACC中，并定义一个label。

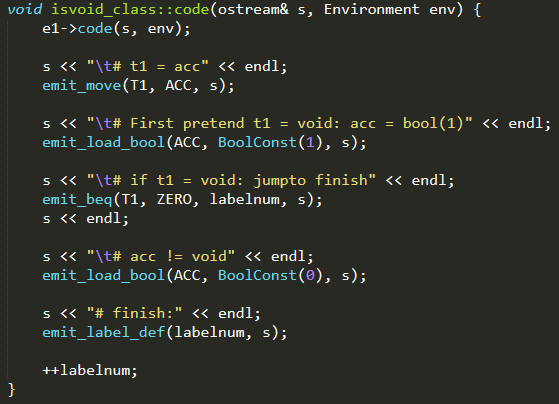


取负的node和判定是否为0的cond则是单目运算，可以理解为单目的四则运算和比较操作。不再详细介绍。

new指令的代码生成涉及堆内存的分配，需要调用object.copy()。每个类型，都必须有一个原型，存在于Data这一区域。我们需要符合原型的规范。在为new生成代码时，首先调用Object.copy()分配空间，其大小正好是一个该类型原型的大小，然后调用该类型的init方法进行初始化。需要注意的是，new的参数可能是SELT\_TYPE。需要进行一定的处理。



is\_void语句的代码生成核心是BEQ。首先假设t1就是void，将1存入ACC。如果t1是void，就jump到finish。ACC保持是1。如果t1不是void，就会执行到将0存入ACC。

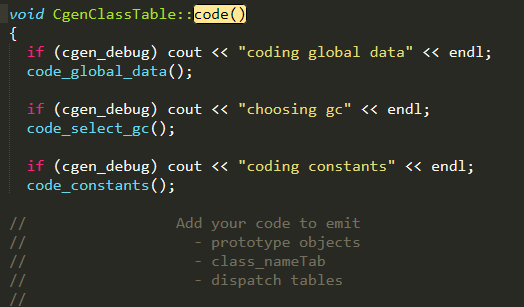


空语句的code也是trivial的。



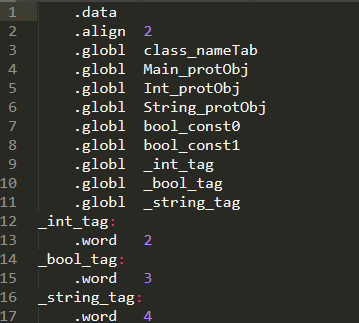
**四、整体代码生成**

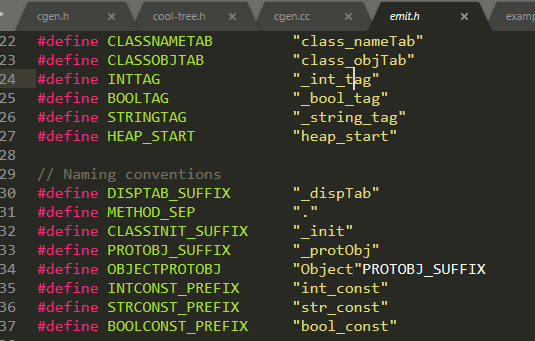
代码生成部分通过CgenClassTable::code()来实现。首先，我们解读一下已经提供给我们的3个函数：code\_global\_data()、code\_select\_gc()以及code\_constants()。



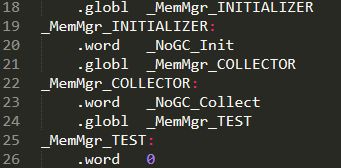
code\_global\_data()做的事情是比较trivial的，其效果可见标准生成器给出的结果。

这一步用到的宏定义在emit.h中，对比即可了解。

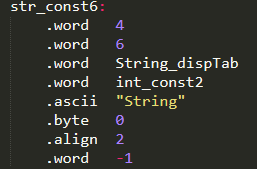
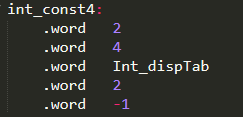




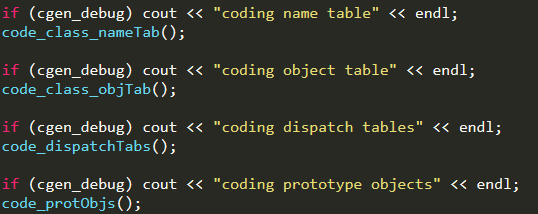
code\_select\_gc产生的代码如下：

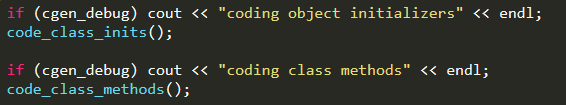


code\_constants()会对stringtable和inttable中的每个常量生成如下代码。

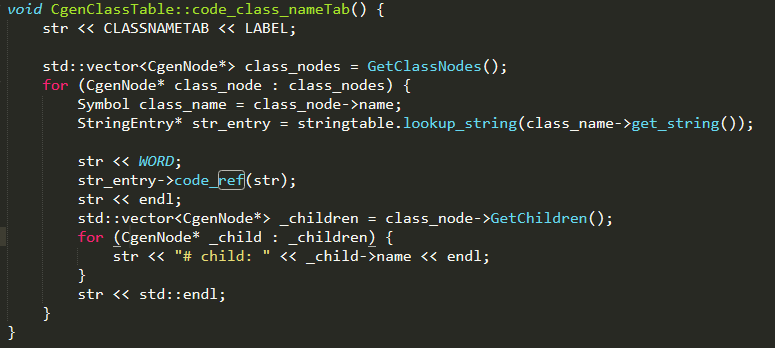
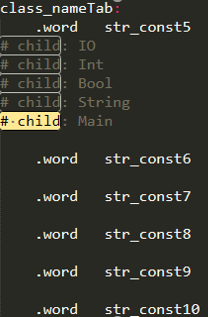
 

接下来，为了完成代码生成这个过程，介绍我们添加的几个函数。

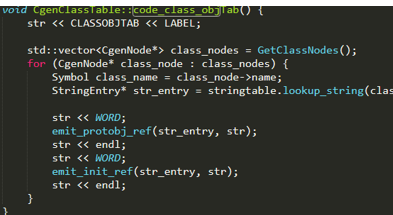
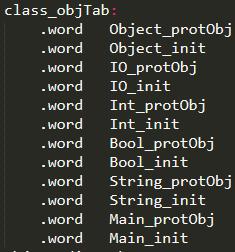




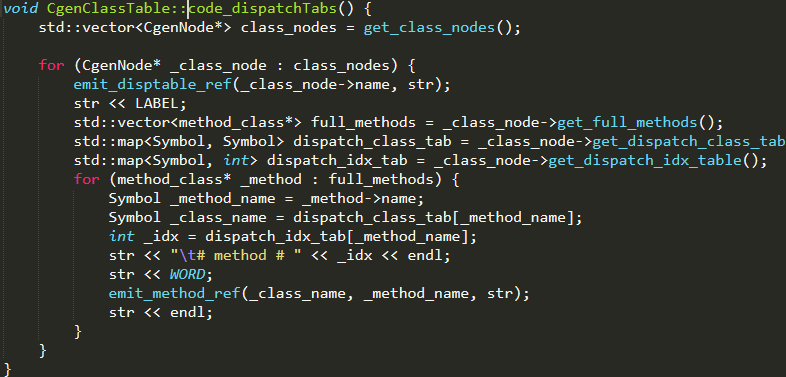
首先使用code\_class\_nameTab函数，为类的名称表生成代码。通过GetClassNodes()函数，可以得到所有类别的节点。然后遍历所有的CgenNode，为每一个类别名查找其在stringtable中的entry，然后生成相关的代码。代码中还包含其子类的注释信息。这一部分是cool要求固定在汇编码中，以便运行时系统查找需要。

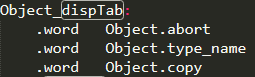
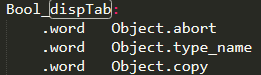
然后调用code\_class\_objTab，为类的对象表生成代码。具体细节和code\_class\_table类似。

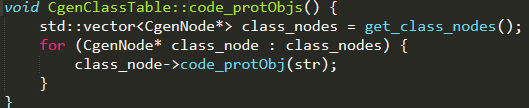
code\_dispatchTabs函数为dispatch表生成代码。该表类似于C++中的虚表，指明每个类下可以调用哪些方法，对于多态的方法具体调用哪一个，涉及动态的类型检查。该表在运行时不会被使用到，但在编译的时候很重要。其中，get\_dispatch\_class\_tab和get\_dispatch\_idx\_table都是访问get\_full\_methods函数中得到的副产物dispatch\_class\_tab和dispatch\_idx\_tab，分别是方法到类名的map和方法到类内向量中所在的下标的map，也就是说可以知道该方法是在哪个（父）类定义的，是第几个方法，这样就可以为他生成代码。



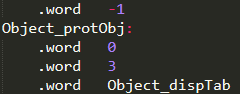
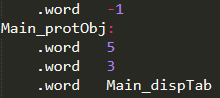
生成的dispatch table形如：

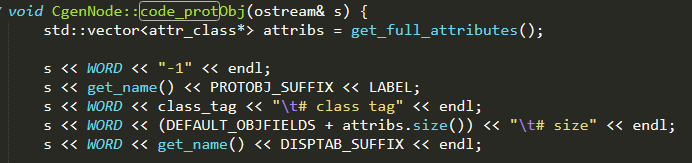
 

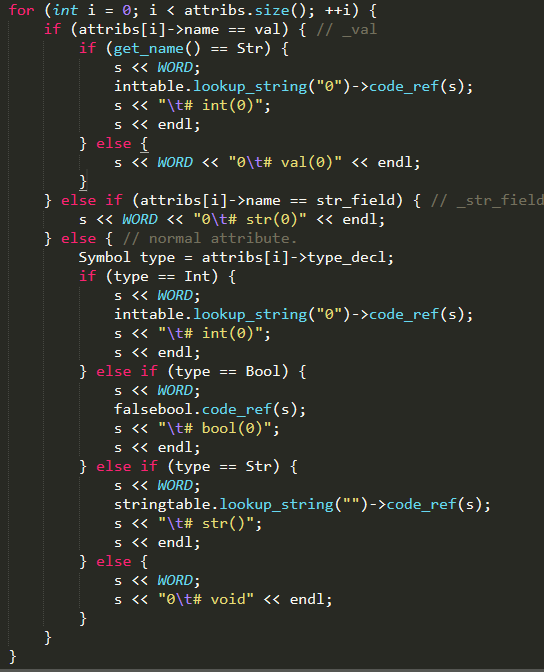
code\_protObjs为每个类的原型对象生成代码，即对每个类结点调用code\_protObj函数。每个类都有一个原型。



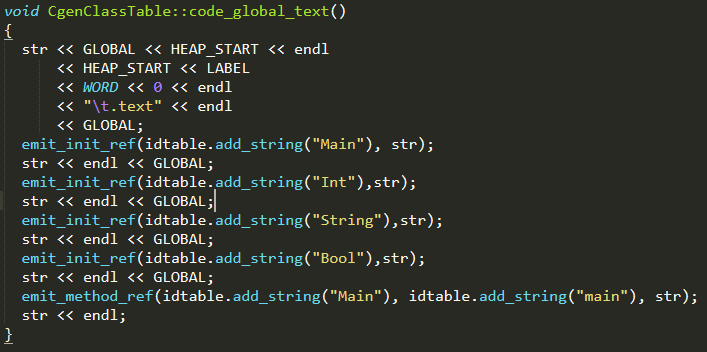
参考标准的生成器给出的格式，实现原型对象生成的代码。调用的方法基本是提供好的emit函数，只要对参考标准输出，对汇编有一定了解，就可以写出来。

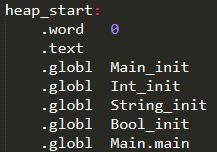
 



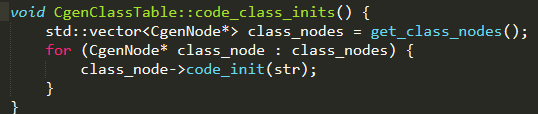


接下来调用code\_global\_text()函数，该函数也是已经提供给我们的。这部分开始定义堆空间的起始位置，以及各基本类型的init方法和Main的main方法声明。

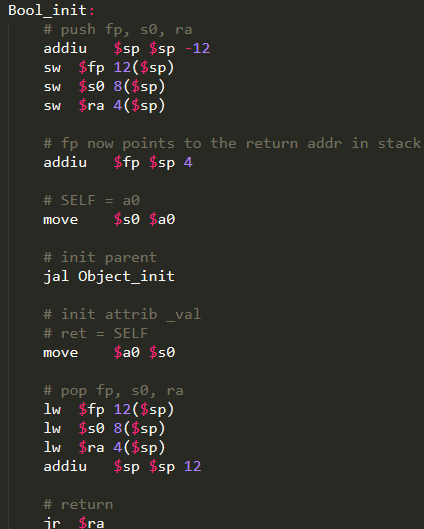




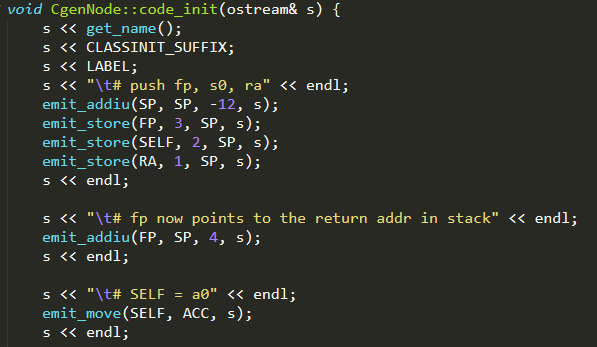
code\_class\_inits()函数为类别的初始化生成代码。



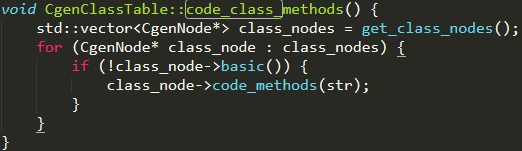
首先，被调用者保存寄存器，创建栈帧，然后，调用父类的初始化，对属性赋予初值，再进行寄存器和栈帧的恢复，返回分配的地址。



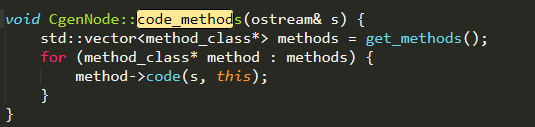
按照这个流程写代码即可。只要掌握了inits函数要干什么、emit函数如何调用，就能写出来。



最后调用code\_class\_methods()函数为非基本类的方法生成代码。

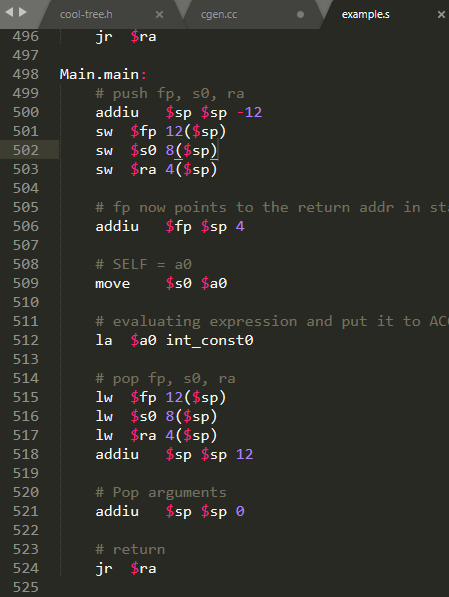


code\_methods将依次调用各method的code方法，即第三部分中已经介绍的代码生成。

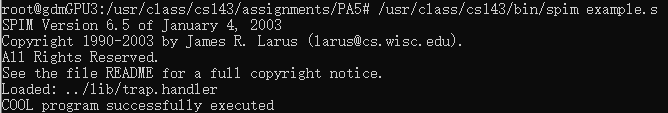


**五、结果**

make dotest



/usr/class/cs143/bin/spim example.s



**六、感想**

PA4好难，我自闭了。这个项目需要对cool、MIPS都有很深的理解，需要结合标准生成器的输出来进行分析，而且需要自己完成的地方很多，不像PA1、PA2比较strateforward，在设计上有很多考量。直接上手太过困难，还是需要对相关知识在事前有充分了解，并参考一下别人的做法。