目标代码生成及模拟

# 一、实验目的

通过上一步中间代码生成的四元式进一步生成最终可以在保护模式下编译的32位汇编代码，并通过汇编指令生成.obj和.exe文件。

# 二、实验环境

系统：WINDOWS7

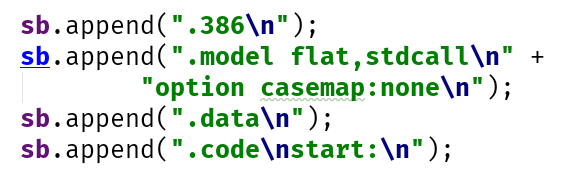
软件：IntelliJ IDEA

# 三、目标代码生成实验步骤

我将目标代码生成分为了如下几个部分：代码开始部分、数据段生成部分、代码段生成部分。文件写入采用的是java.io.FileOutputStream，通过将所有要写的东西加入一个StringBuffer类的串中进行写入，StringBuffer与String相比的优点是可以随时对StringBuffer通过append、insert等方法进行修改，而String在初始化后它的值就不能被修改了。

## 3.1 代码开始部分

代码开始部分就是最开始的一系列声明，是对于任意四元式均需要生成的部分，例如使用模式、区分大小写等等，这部分我直接加到了文件中。



## 3.2 数据段生成部分

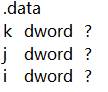
在这里首先要获取四元式中的Element，首先定位到function，然后获取它的ElementList



使用foreach对每个四元式进行判断，由于我在生成四元式的时候直接将定义类语句的op设为了int/char/…，因此直接用正则表达式进行判断



如果match，那么直接插入到.data后面，注意这里是用insert方法而不是append，因为已经添加过了.code，而且如果在代码中又出现了数据定义的部分，那么都需要加入到数据段中。由于是倒序插入，在汇编代码中定义的变量顺序会与C代码中变量定义的顺序相反

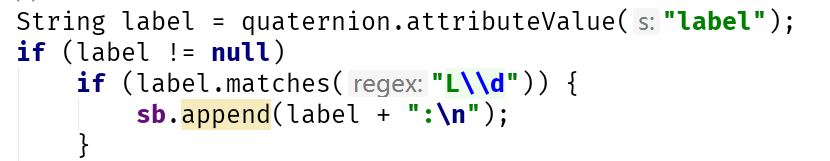
C: ASM: 

## 3.3 代码段生成部分

代码段的内容可以分为标签、赋值操作、比较大小操作、条件转移、非条件转移（直接转移）、加减操作

### 3.3.1 标签

对于每个四元式，都要判断它有没有label，如果有label就先append它：



### 3.3.2 赋值操作

我们知道，对于C代码的a=b，汇编代码为mov eax,b mov a,eax 因为两个变量不能直接进行操作，必须要寄存器过渡一下，因此对于一个赋值四元式，应该生成两行asm代码。当然，这里有一点问题，在后面会解决。

### 3.3.3 比较大小操作

比较大小在汇编中有test和cmp两种指令，在这里选用cmp指令。由于比较大小操作涉及到2个四元式（比较和条件转移），因此需要将两个四元式连接起来一起看，对于第一个四元式，例如我代码中的

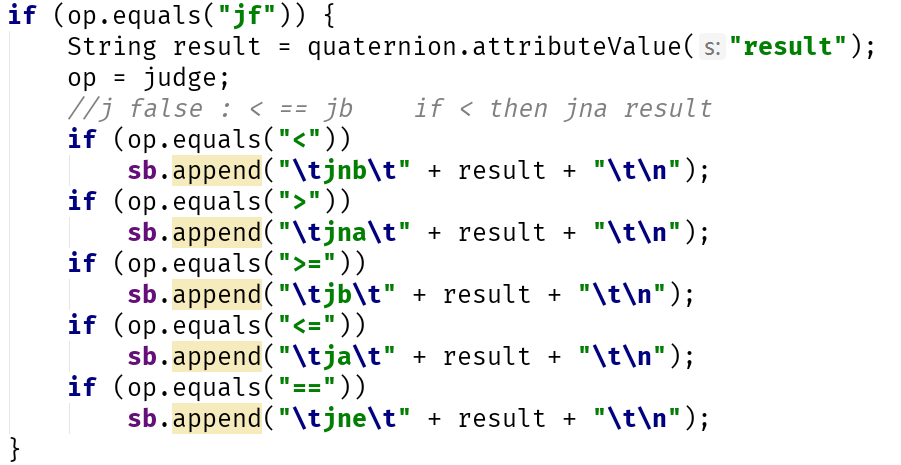


可以生成3行代码，注意这里i和10尽管10可以直接进行cmp，但是i是变量，不可以直接cmp，因此要放到寄存器中，但是如果比较的是i和j，那么都需要放到寄存器中，为了统一，不论是不是变量，我将两个参数全部放入寄存器中，这样尽管代码多了一行，mov指令并不会对整个程序造成很大的影响。

在生成第一行四元式的代码后，就需要进行条件转移了，那么一定要保存这个条件，是大于还是小于还是什么，因此我用一个String变量来保存，然后continue到下一条四元式，进行条件转移

### 3.3.4 条件转移

上面以及保存了转移的条件，那么当op为jf的时候我们就需要对条件进行判断，注意是不满足转移，因此需要对保存的条件取反，例如小于，那么对应的就是jnb：



### 3.3.5 直接转移

直接转移j对应的就是jmp指令，因此直接append就可以了。

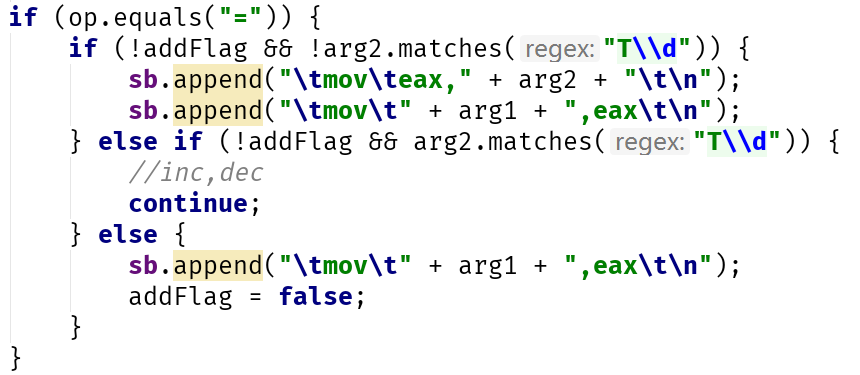
### 3.3.6 加减操作

在四元式中，无论是自增、自减还是普通的加减操作，都是通过两行四元式完成的，例如我中间代码中的：



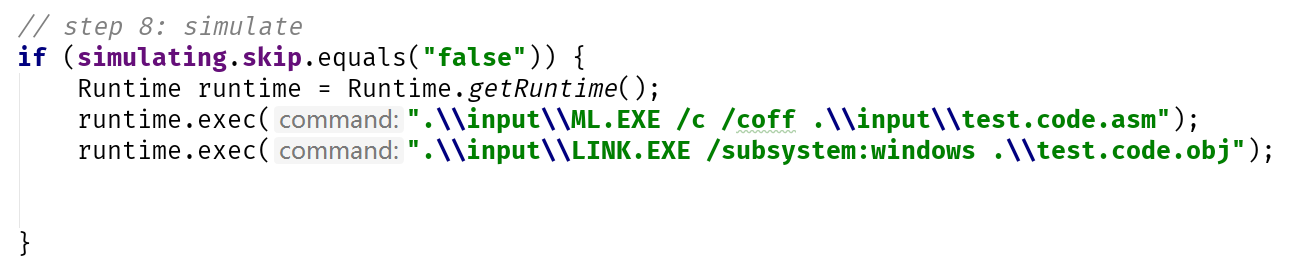
但是，对于arg2为1的时候，汇编中有对应的inc、dec操作，因此在这里要考虑，具体实现在代码中有体现。

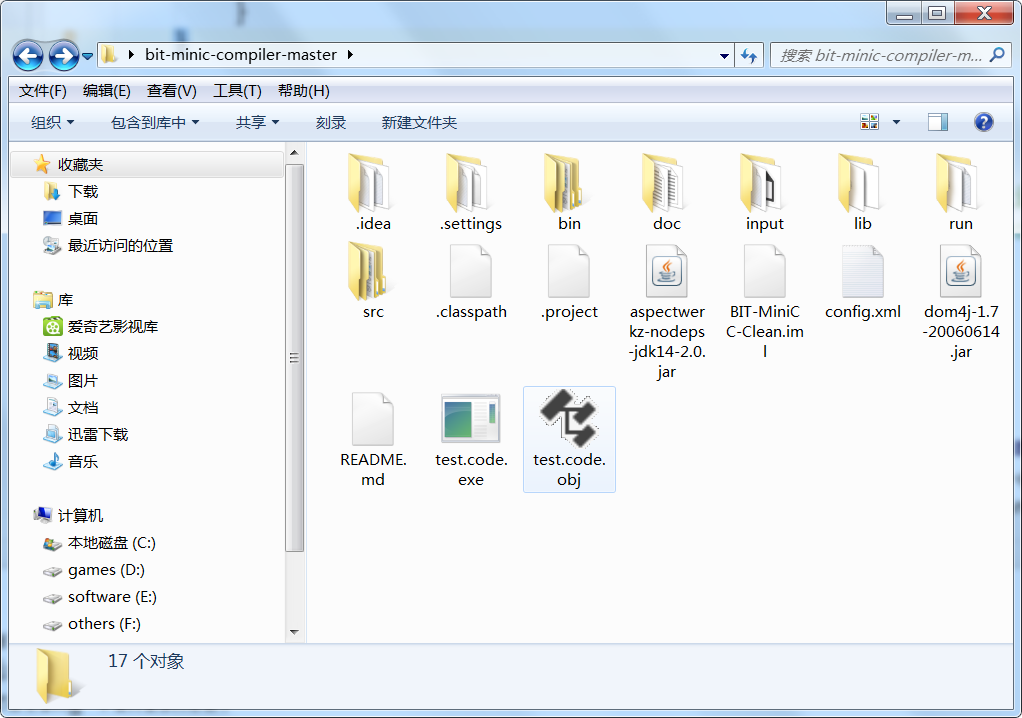
那么就会产生一个问题，第一行四元式的T3，第二行的T3应该如何保存呢？因为第二行是赋值语句，包括第一行的加法都不能对两个变量直接进行操作，因此仍然需要寄存器来保存他们，并且如果为inc、dec的话，下面那行的变量赋值就可以无视。



# 四、模拟步骤

在模拟的步骤中，采用了java的runtime方法，调用exe文件对asm进行汇编，由于当前目录是项目的根目录，因此最后产生的obj和exe文件都在根目录下。注意生成的exe文件只能在32位下运行。

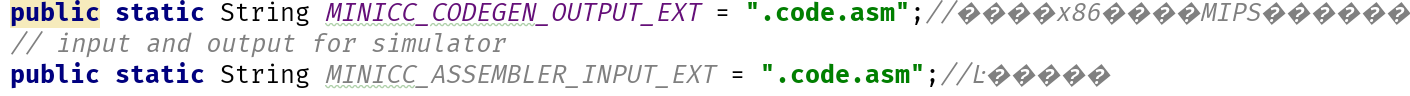




# 五、实验总结

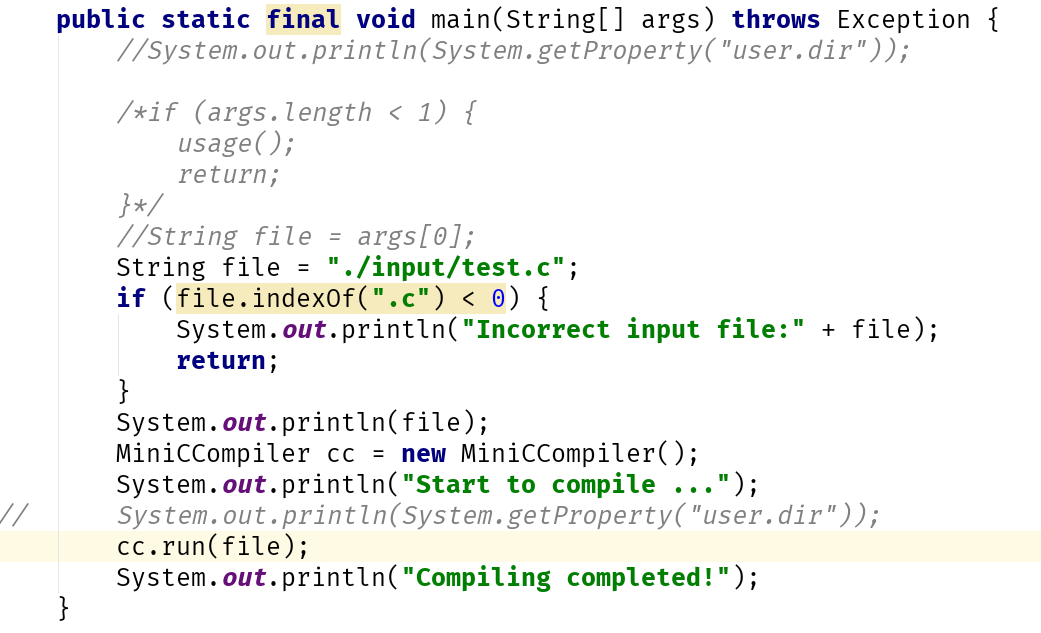
## 5.1 说明

首先，生成的为WIN32保护模式的汇编代码，后缀改为了.asm



对于代码test.c，生成的语法树为test.tree.xml，生成的中间代码（四元式）为test.ic.xml，生产的目标代码为test.code.asm

我对运行方式进行了修改，不需要像原来一样输入文件路径，默认路径为./input/test.c



并且将依赖的jar包放在了./lib下，因此直接build后run就可以了。

## 4.2 心得体会

在我看来，生成目标代码的过程完全是依赖于中间代码生成的，在中间代码生成的基础上，做的仅仅是xml到.asm的一个转换，因此完成的相对中间代码生成来说比较轻松，做目标代码生成给我最大的收获就是学习了更多关于正则表达式的东西以及java的文件操作。

在模拟的部分，我本来是想要用python脚本实现，但是java调用python脚本的时候总是提示找不到module，试过了很多方法，怎么都无法解决，因此直接用java自己的调用.exe。

经过了一个学期的努力，终于写出了属于自己的编译器，词法分析、语法分析、中间代码生成、目标代码生成、模拟，都是用的自己的代码。尽管这个短短一千多行的代码组成的编译器支持的代码很有限，但是还是很满足，我觉得我真的算是完全理解了编译器处理源代码的各个环节，并且对具体实现以及其中可能遇到的问题有了更多的认识。此外，我对于Java语言也是使用的更加熟练了。

我认为整个环节中最难的两个地方是语法分析和中间代码生成，这两个部分也是我完成耗时最长的两个部分，而他们很难完成的原因就是对于理论知识掌握的不熟，比如语法分析，我在最开始的时候甚至都没有想到要用书上的LL1或者LR文法完成，只是自己去琢磨，发现越来越复杂，后来认真的看完书后，发现其实原理是很清晰的。

最后感谢老师以及学长为搭建bit-minicc所做出的努力以及上课时的教导，让我能够对编译原理有更加深入的认识，在这个框架的基础上做出了自己的编译器。