**编译原理与设计**

**实验报告**

实验名称： Lab 7：中间代码生成实验

姓名/学号： 李昊阳/1120203053

1. **实验目的和内容**

**实验目的：**

（1）了解编译器中间代码表示形式和方法；

（2）掌握中间代码生成的相关技术和方法，设计并实现针对某种中间代码的编译器模块；

（3）掌握编译器从前端到后端各个模块的工作原理，中间代码生成模块与其他模块之间的交互过程。

**实验内容：**

以自行完成的语义分析阶段的抽象语法树为输入，或者以 BIT-MiniCC 的语义分析阶段的抽象语法树为输入，针对不同的语句类型，将其翻译为中间代码序列。

具体步骤如实验指导文档所示。

1. **实验环境**

设备：RedmiBook 14 锐龙版

操作系统：Windows 10 Pro, 64-bit (Build 19045.2604) 10.0.19045

Java：jdk-17

Java IDE：IntelliJ IDEA 2022.2.5 (Community Edition)

Antlr：ANTLRv4 4.11.1

1. **实现的具体过程和步骤**

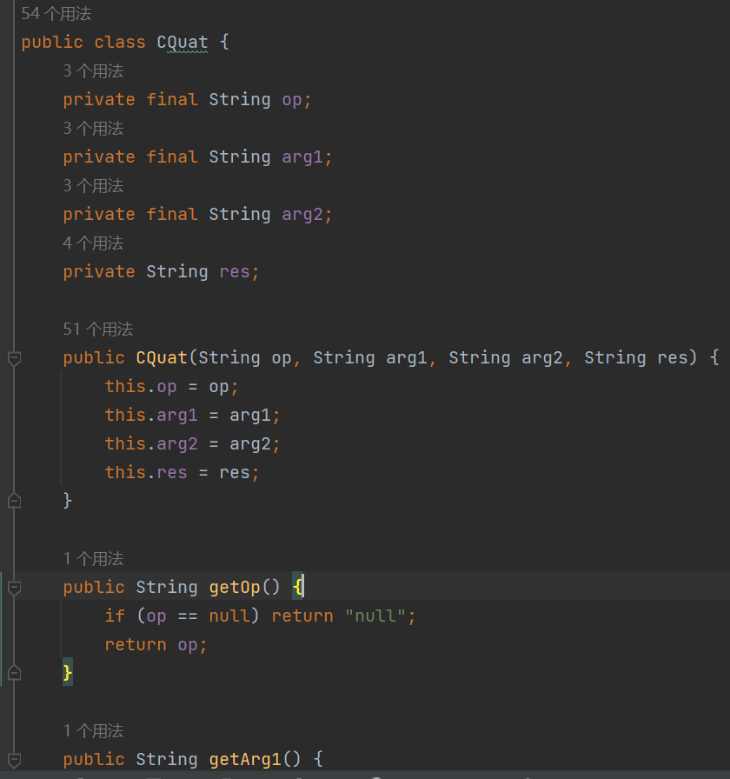
按照实验文档的指示，在前置实验：语法分析实验与语义分析实验的基础上，以生成的抽象语法树为输入，针对不同的语句类型，将其翻译为中间代码序列。

因此，我依旧使用Antlr来生成抽象语法树，并在遍历抽象语法树的时候同时进行语义检查与生成中间代码工作。本次实验只需在原有lab6的基础上进行工作。

因为四元式形式和x86汇编形式相差较多，为了相对简便的进行目标代码生成，因此中间代码构造时选择与x86汇编相近的形式。在翻译控制语句时，如选择、循环、跳转等语句，使用较简略的jump语句，减少对代码的翻译。

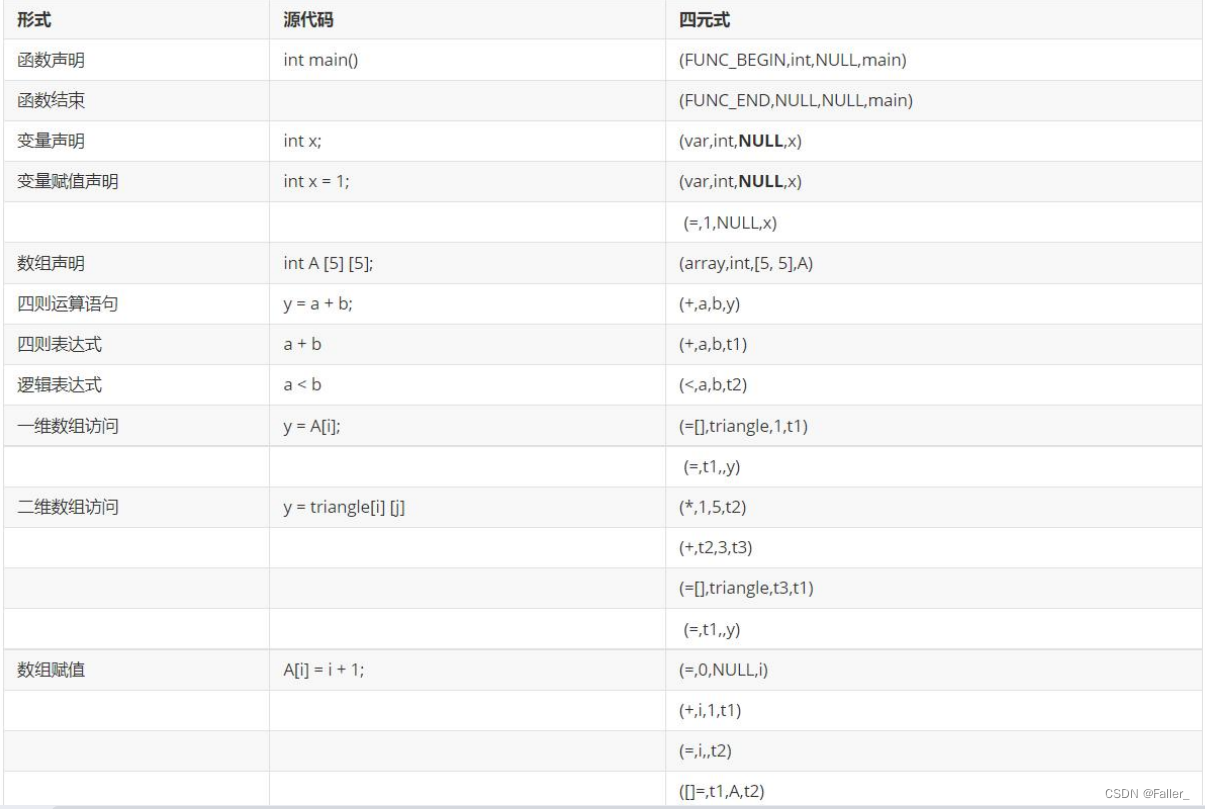
1. 定义四元式与四元式类CQuat

采用（operator, operand1, operand2, result）的四元式基本形式。其中：op为一个二元（也可以是零元或一元）运算符或控制信息；arg1和arg2为两个运算对象，可以是变量、常数或者系统定义的临时变量名，也可为null；result为运算结果。

CQuat定义如下：

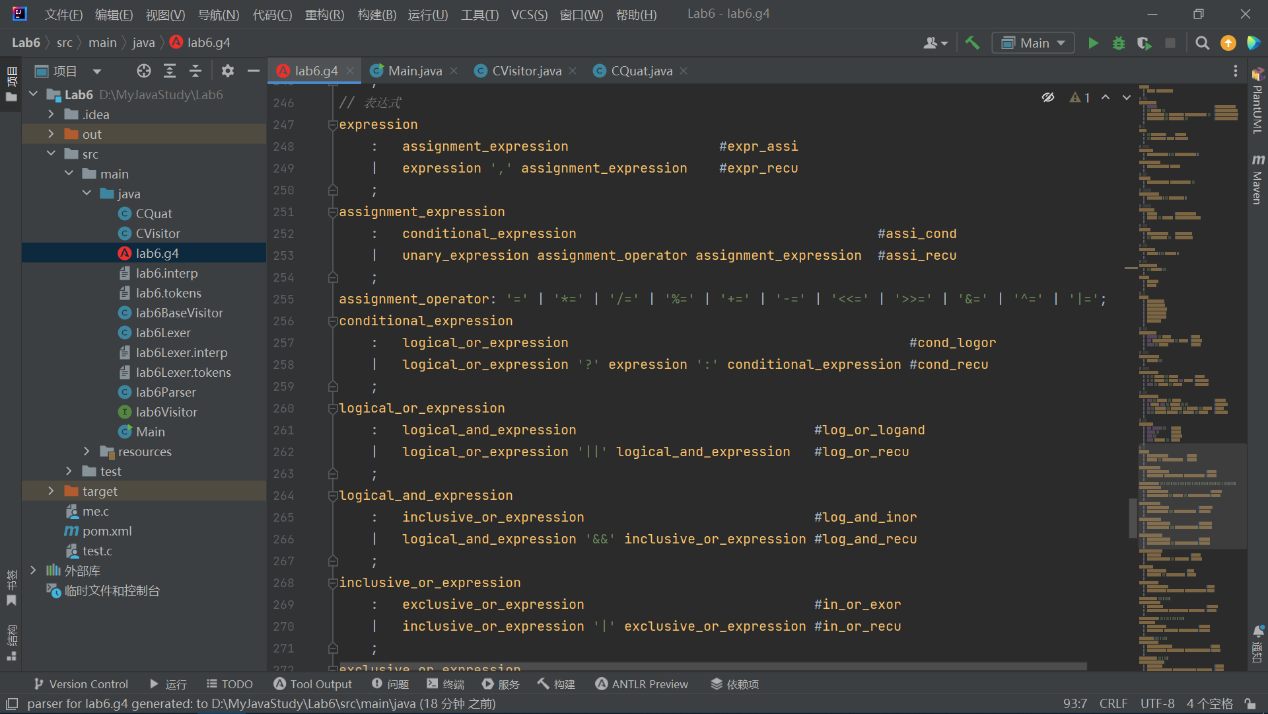
包括op、arg1、arg2、res四个字段，由String类型储存，通用性最好。以及一些必要的公有方法。

四元式的编写规则大致遵守如下：

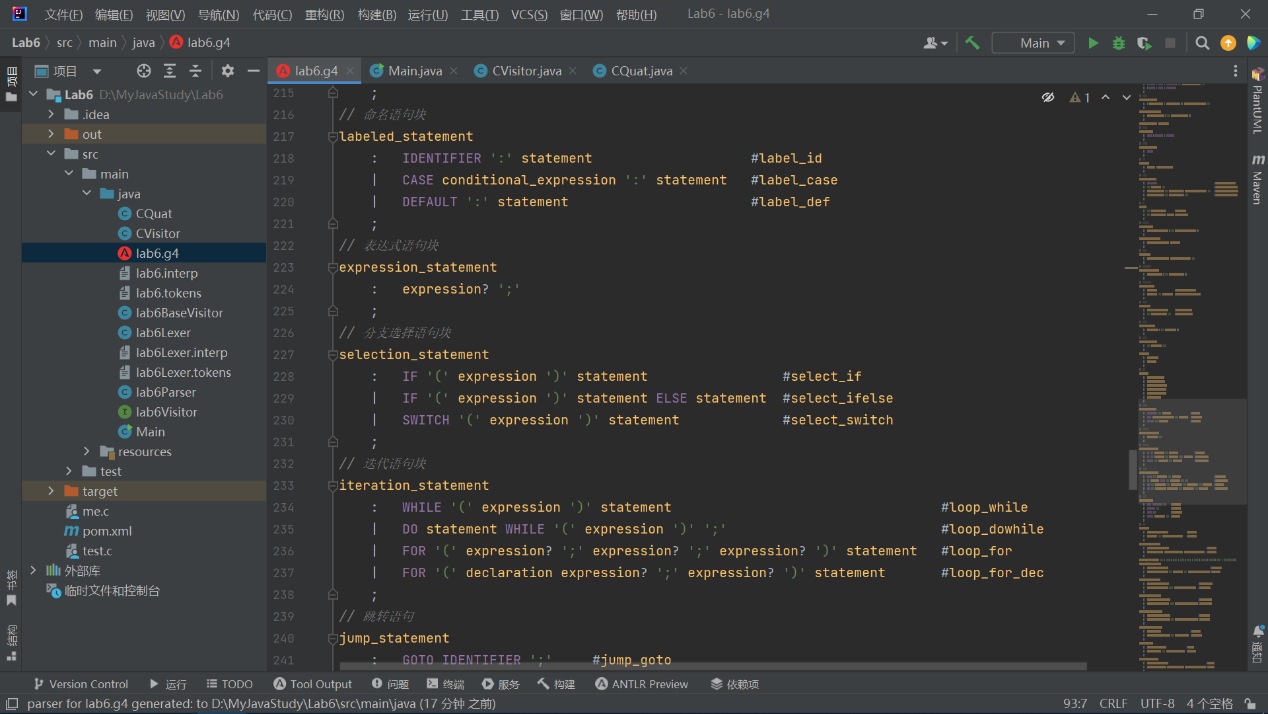
在此基础上加以修改，增加了控制流，把选择、循环、跳转语句以真实的控制流表现出来，去掉了上表中.if、.while四元式，增加了多种jump四元式。

1. 修改lab7.g4文件

此前实验的所编写的文法已较为完备，但由于我将所有直接递归语法修改为了一次性匹配，导致在后续编写visit函数时遇到了困难。因为一次性匹配不方便进行函数的递归调用，必须在一次调用的循环中解决完整个句子，对于符号的结合次序、四元式的生成会造成困难。因此将大部分expression语法回改为递归定义，但对原文法中冗余的文法生成与部分递归的修改予以保留。

如下所示：

因为Antlrv4版本可以消除直接左递归，而无法消除间接左递归，在修改文法时应多加注意。对于有多个备选分支的语法规则，我们可以为各个备选分支添加一个标签，只需要在备选分支的右侧，以#开头，后面跟上任意的标识符即可。标识符全局唯一。如果备选分支上面没有标签，Antlr只会为每条规则生成一个访问方法，相反的，Antlr会为每条分支都生成一个访问方法。需要注意的是，为一个规则的备选分支添加标签时，要么全部添加，要么全部不添加。

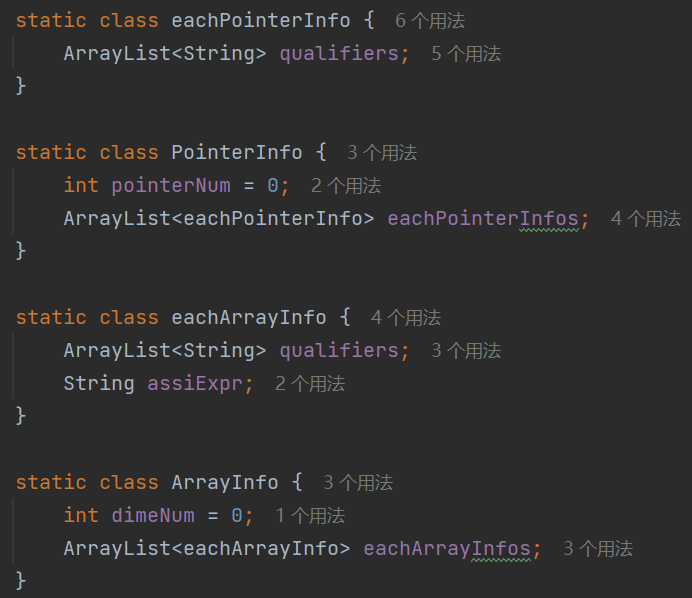
如下所示：

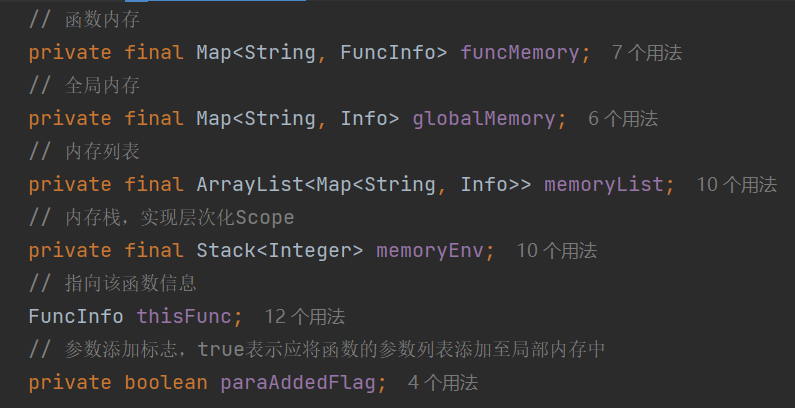
为语句的备选分支添加标签，可以更好地处理不同的语法规则，编写不同的visit函数。

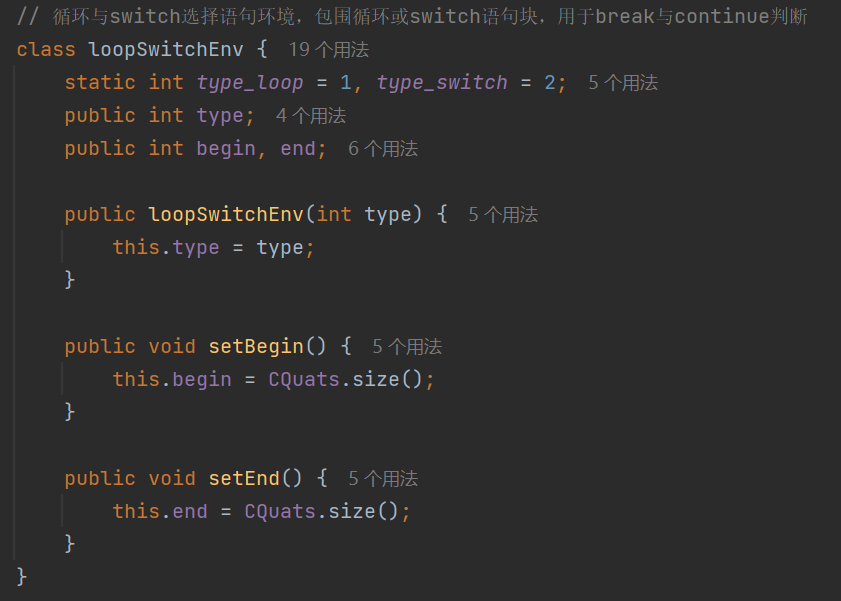
其余语法详见lab7.g4。

1. 修改CVisitor类

在lab6的CVisitor类的基础上，进行修改与增添，使其在访问语法树时同时进行语义分析与中间代码生成。新的CVisitor类是继承自lab7.g4文法生成的lab7BaseVisitor<T>泛型类，修改泛型为String，以匹配四元式类型。

在CVisitor类中，增加了静态类PointerInfo、ArrayInfo，用来保存变量的指针信息与数组信息，Info类、FuncInfo类同时拥有PointerInfo成员与ArrayInfo成员。

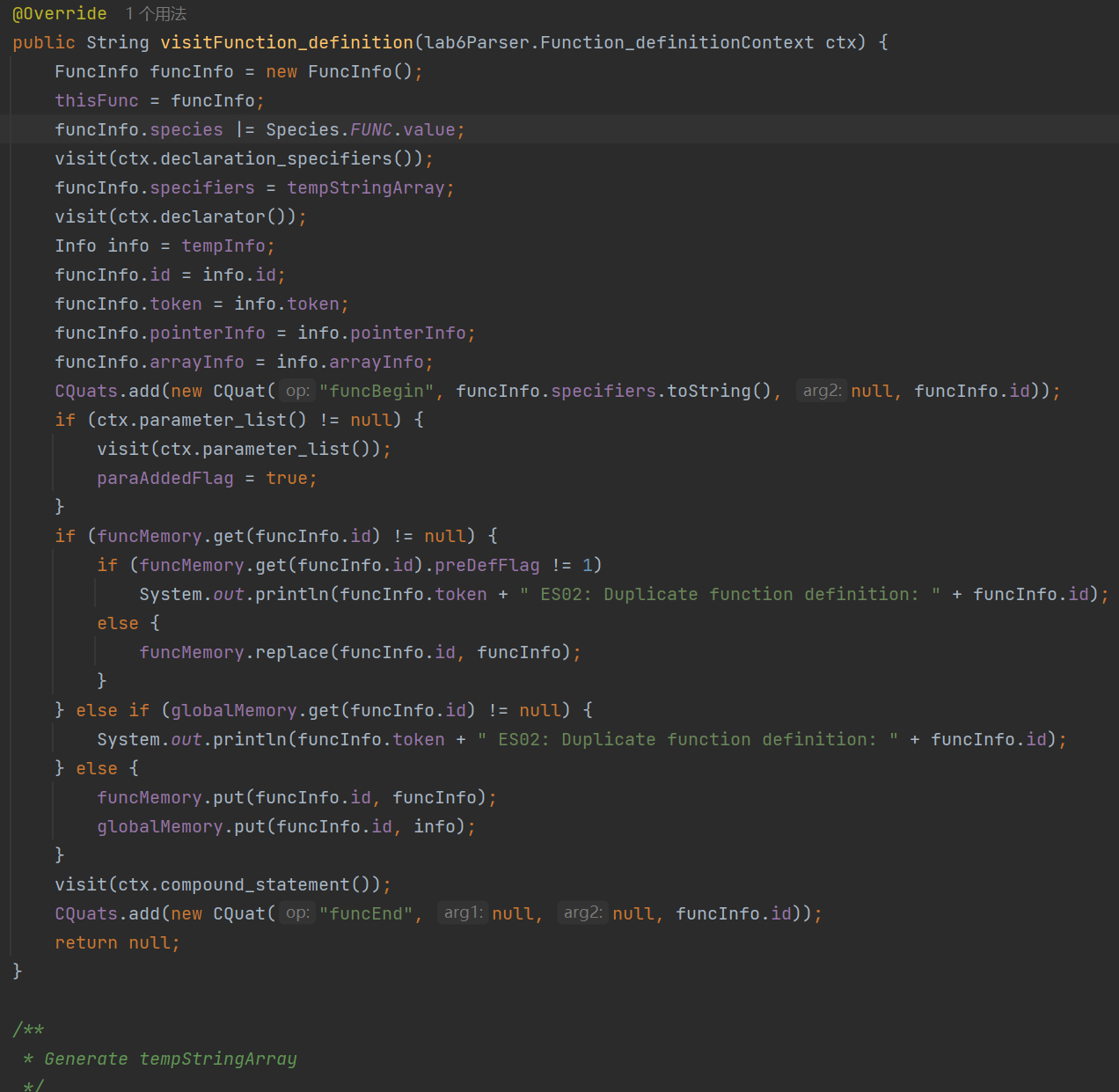
CVisitor类中还定义了大量全局变量，辅助完成语义检查与中间代码生成功能。以下为定义与变量用途的注释：

还定义了loopSwitchEnv内部类，描述循环或switch选择语句块，辅助break与continue语句（若有）进行跳转：

接着重写大部分visit函数，以实现遍历语法树，同时进行语义检查与中间代码生成的功能。

下面以几个重要的visit函数为例：

* 1. visitFunction\_definition

调用该函数时，会访问上下文（ctx）指向的函数结点。

先实例化一个FuncInfo类，将thisFunc指向该funcInfo。

由于visit()是其祖先的抽象方法，调用visit()会自动调用与参数上下文指向的结点对应类型的visit函数，此例中等同于调用visitDeclaration\_specifiers函数，因此后续均将使用visit简化编写。

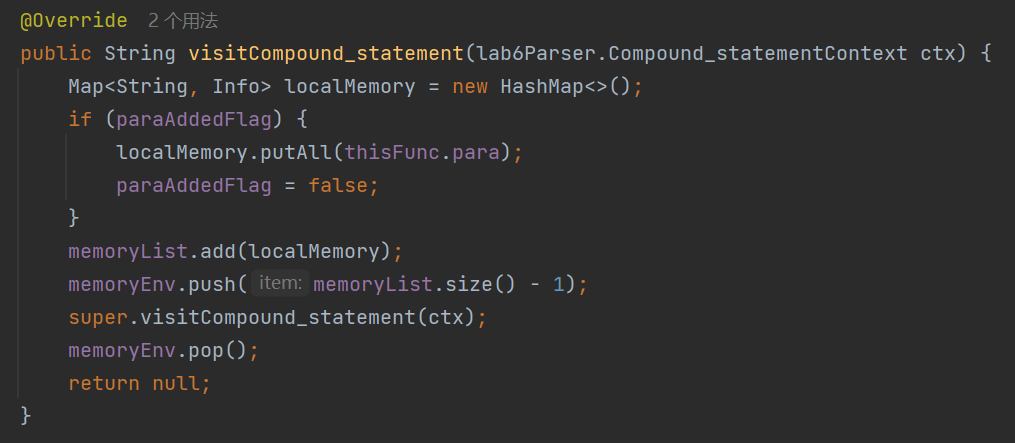
访问函数的修饰符declaration\_specifiers，将修饰符属性写入funcInfo中。然后访问函数的声明符declarator，确定函数ID以及指针、数组信息，并保存声明的token，便于检查时报错。

若函数拥有参数列表，则访问参数列表parameter\_list，将参数保存在funcInfo的para符号表中，便于查询，并讲paraAddedFlag置true。

在将funcInfo放入全局函数符号表前，需确定是否有重名函数或变量。若函数已声明过但未定义，则修改定义；若有重名变量或函数，则输出ES02错误信息，表示函数重定义，并指出重定义标识符的token信息；否则讲函数定义添加至函数内存与全局内存中，避免变量与函数重名。

最后访问其函数主体compound\_statement。需注意使用funcBegin与funcEnd四元式环绕，表示函数定义范围。

* 1. visitCompound\_statement

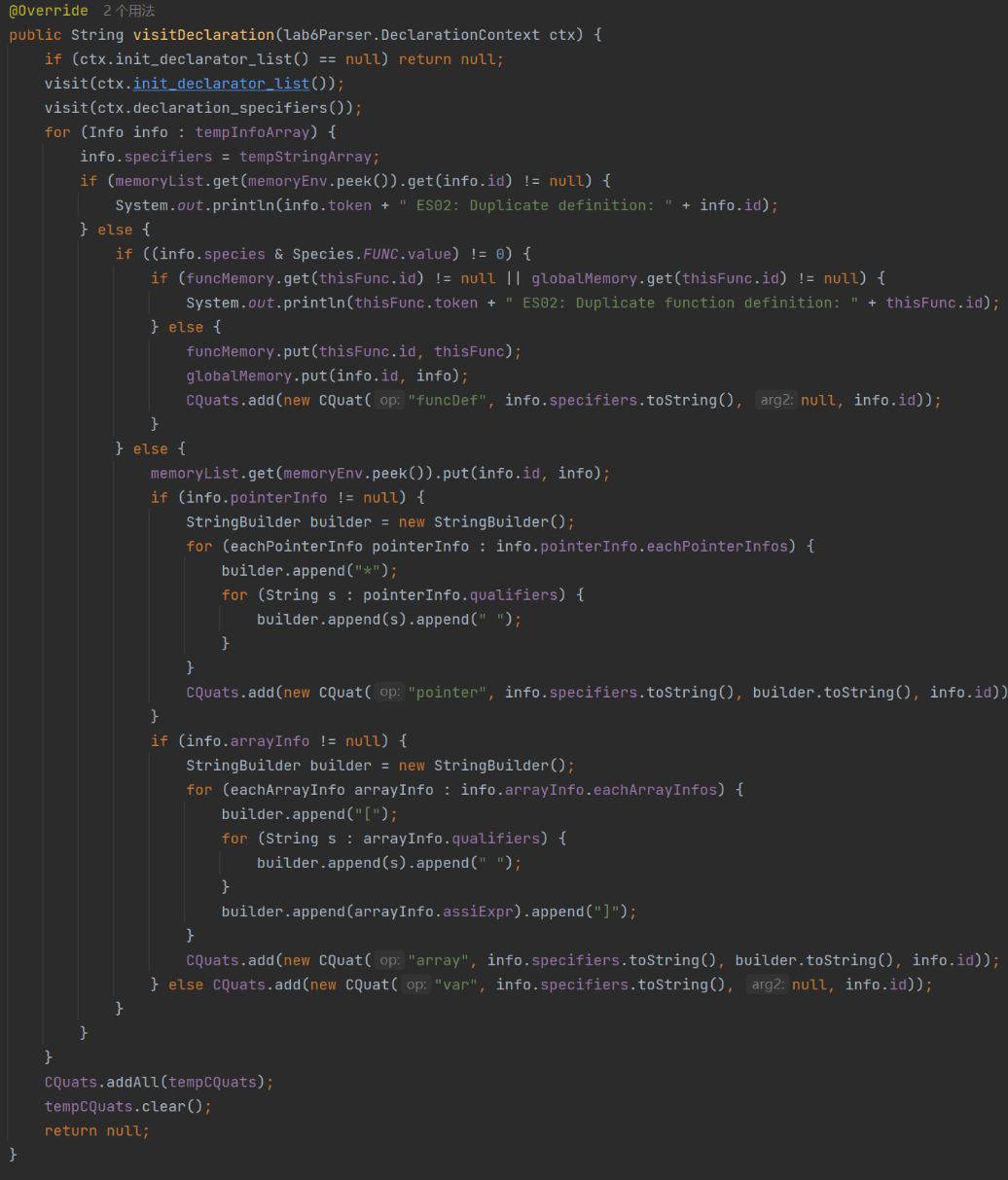
调用该函数时，会访问上下文指向的复合语句块。

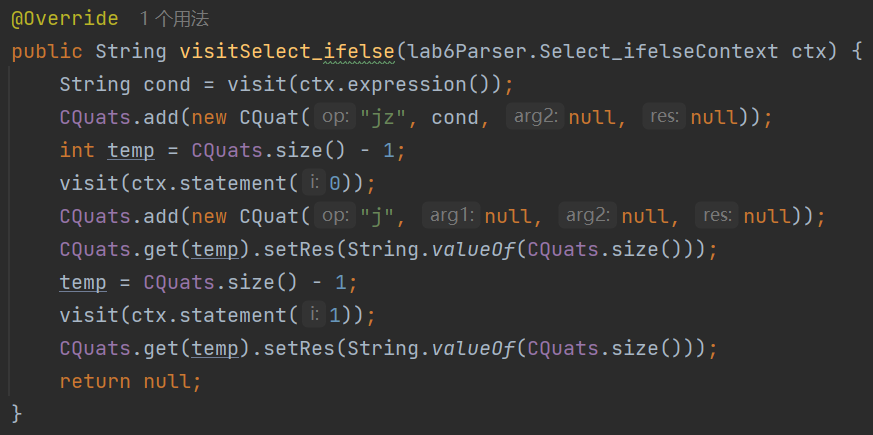
因为C语言具有Scope属性，对于每一个复合语句块，其内层Scope里面的符号对外层是不可见的。因此需单独创建一个局部内存localMemory。若该复合语句为函数主体，则将para添加至局部符号表中，便于后续查询。

将局部内存储存至内存列表中，并压至内存栈中，以实现层次化Scope，访问内部结点结束后，内存栈弹出。

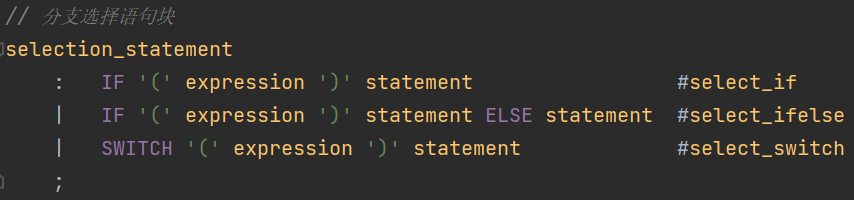
* 1. visitDeclaration

调用该函数时，会访问上下文指向的声明。

该函数会对于声明中的每个变量都实例化一个Info，对于每个Info，会判断其是否是函数定义，再分别查询内存中有无同名变量，若有，则输出ES02错误信息，表示变量重定义，并给出对应变量标识符的token信息。同时，对于不同类型的变量，如指针、数组或普通变量，生成不同的四元式，如pointer、array、var等。最后将初始化赋值的中间代码加入CQuats末尾。

* 1. visitSelect\_ifelse

调用该函数时，会访问上下文指向的ifelse选择语句。

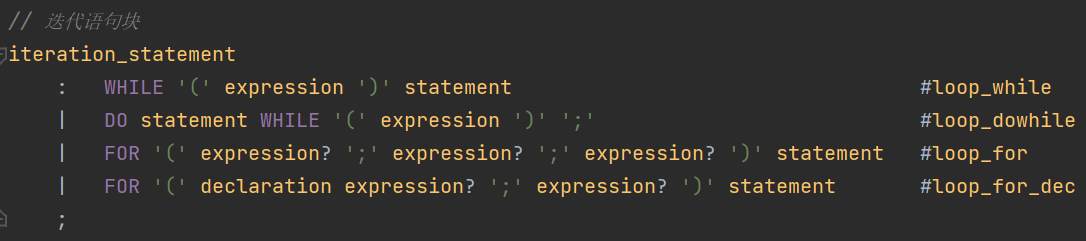
该语句是selection\_statement语句的备选分支，函数为单独的visit函数，其上下文Select\_ifelseContext是Selection\_statementContext的子类。

在该函数的控制流中，会首先访问条件语句，并根据条件作出跳转，即jz四元式，若条件为0，则跳过if语句块，进入else语句；否则反之。但是jz四元式中要跳转的位置尚不清楚，因为还未访问后面的语句块，因此只能先置null，并记录该句位置。

然后访问if后的语句块，写入j四元式，以跳过后面的else语句块。此时我们已知晓第一个jz将要跳转的位置，即当前四元式列表中最后一个元素的后一个位置。修改jz四元式后，同样的，j四元式我们也不知晓跳转位置，再次记录位置。访问else语句块，得到需跳转位置后，修改j四元式。

* 1. visitLoop\_while

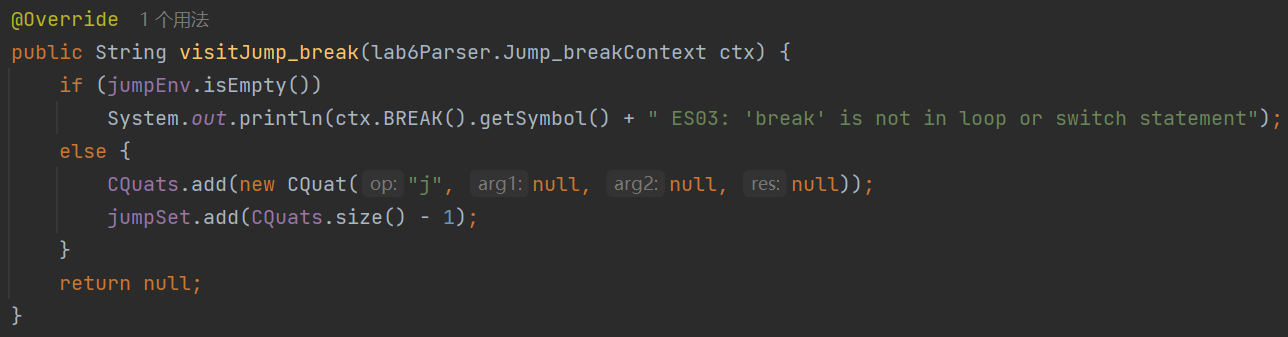
调用该函数时，会访问上下文指向的while循环语句。

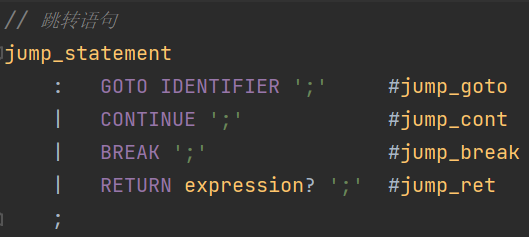
该语句是iteration\_statement语句的备选分支，函数为单独的visit函数，其上下文Loop\_whileContext是Iteration\_statementContext的子类。

首先使用loopSwitchEnv环境包围，并添加置jumpEnv中，若循环内部有continue则跳转至环境首部，若有break则跳转至环境尾部。设置首部位置后，访问条件语句，结果记录在cond中。添加jz四元式，若条件为0，则跳出循环。访问循环体，添加j四元式，返回至环境首部以形成循环。最后设置尾部位置。

需注意的是，尾部位置在最后才会确认，因此jz与break语句（若有）需重新设置跳转位置，方法与上述函数相同。

* 1. visitJump\_statement

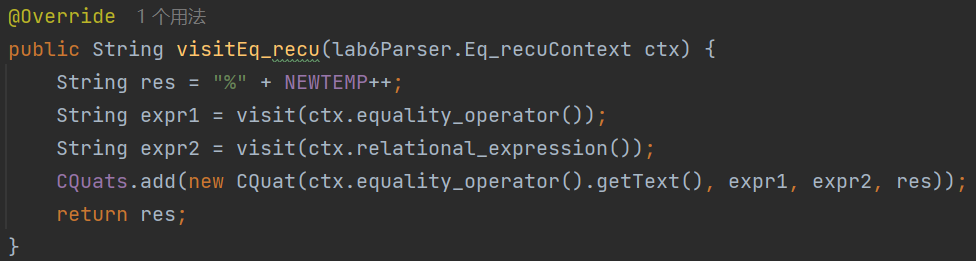
调用此函数时，会访问上下文指向的break跳转语句。

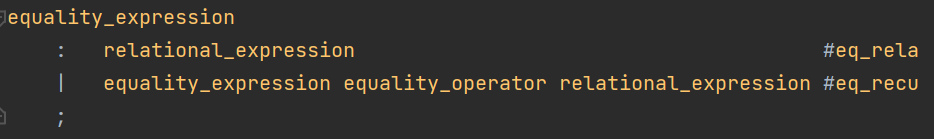
该语句是jump\_statement语句的备选分支，函数为单独的visit函数，其上下文Jump\_breakContext是Jump\_statementContext的子类。

在访问break语句时，会查询是否被jumpEnv包围，若为空，则表明break既不在循环中，也不在switch中，无法使用break。因此输出ES03错误信息，指出该break对应终结符的token信息。

若不为空，则添加j四元式，并将此句添加至待设置的跳转集jumpSet。

* 1. visitEq\_recu

调用该函数时，会访问上下文指向的相等表达式的递归分支。

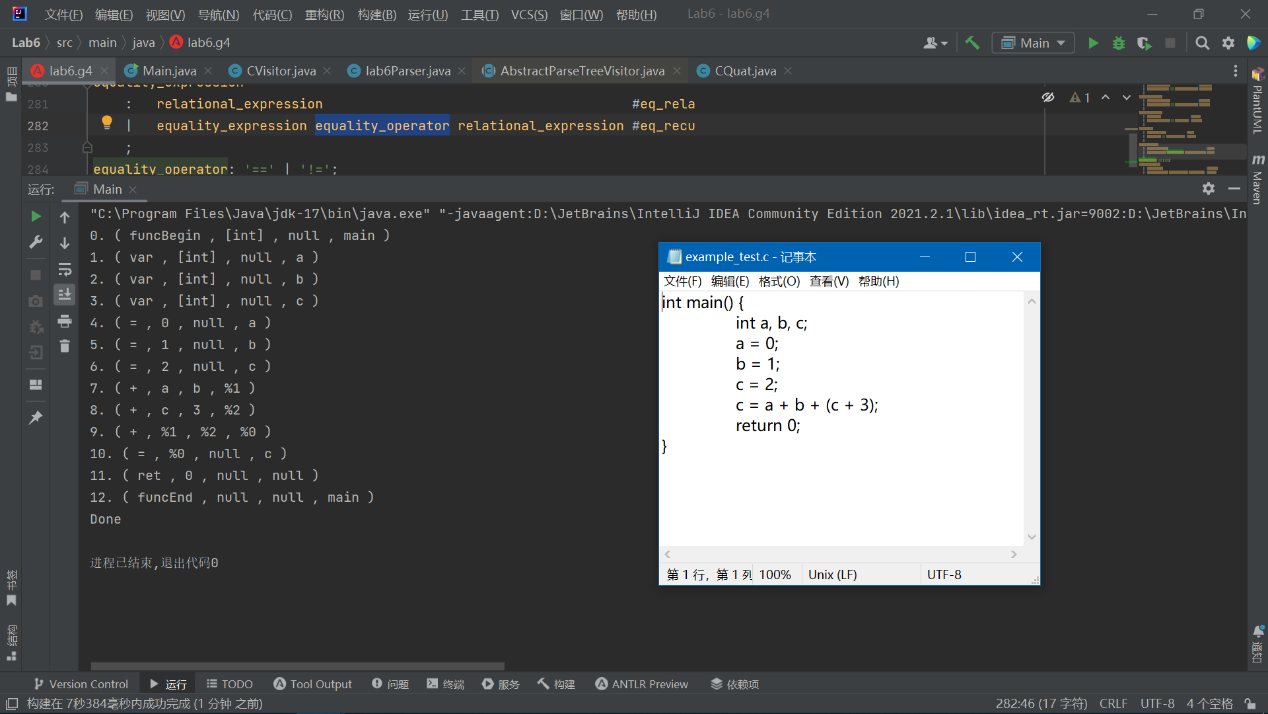
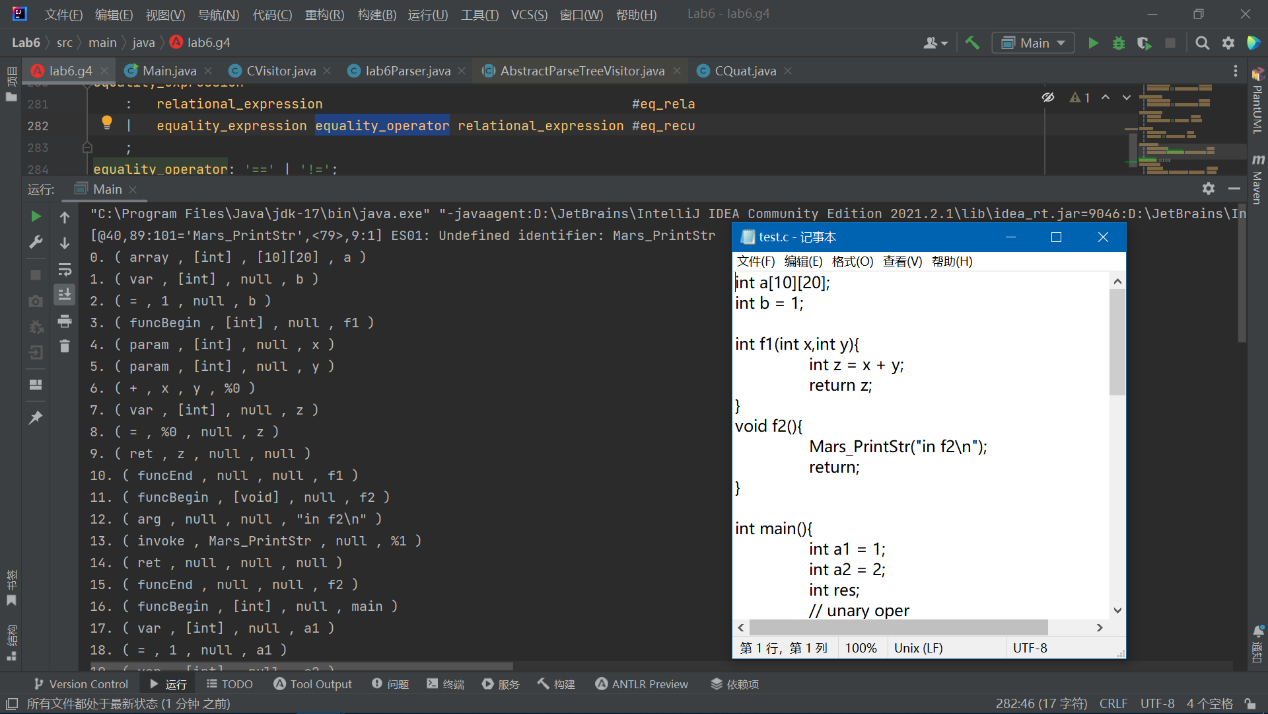
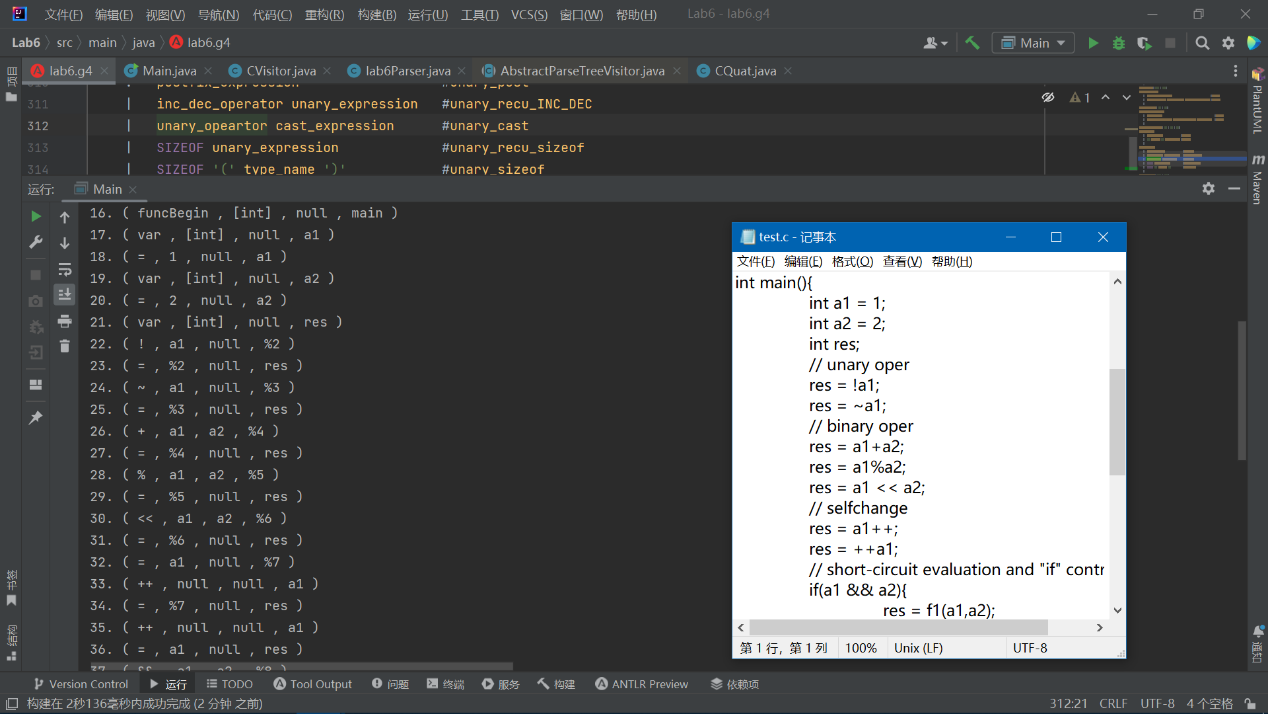
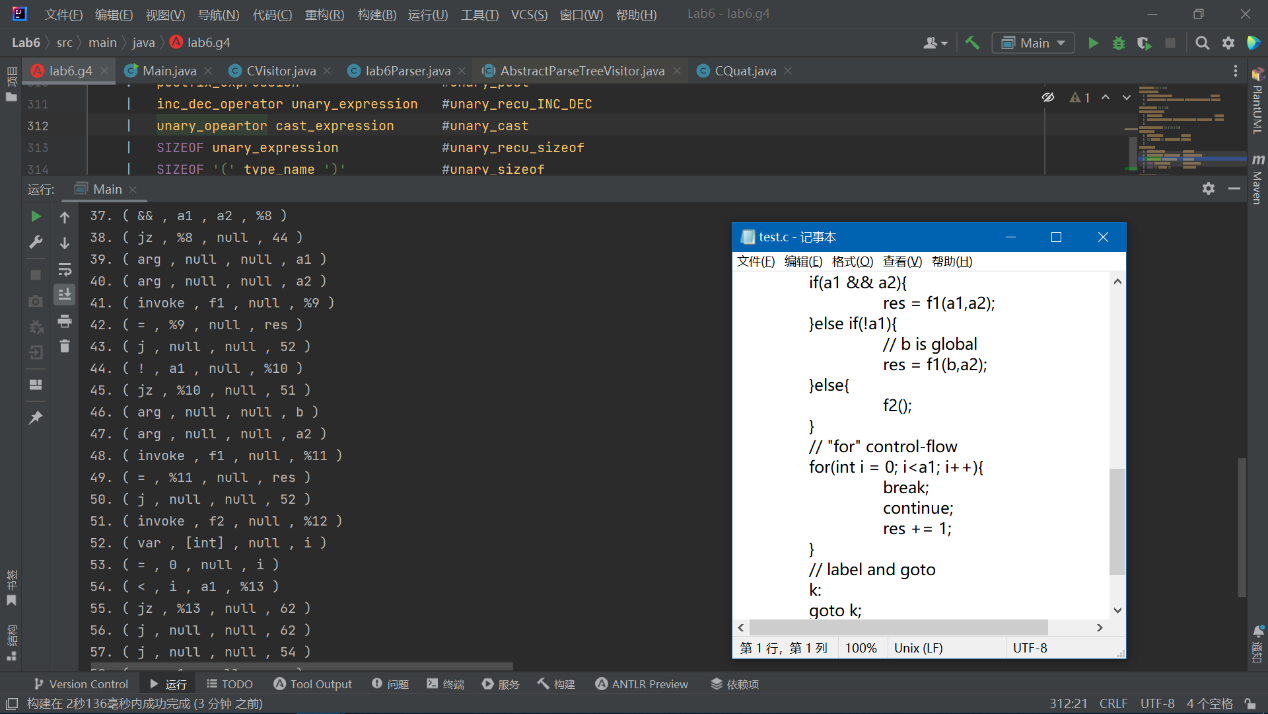
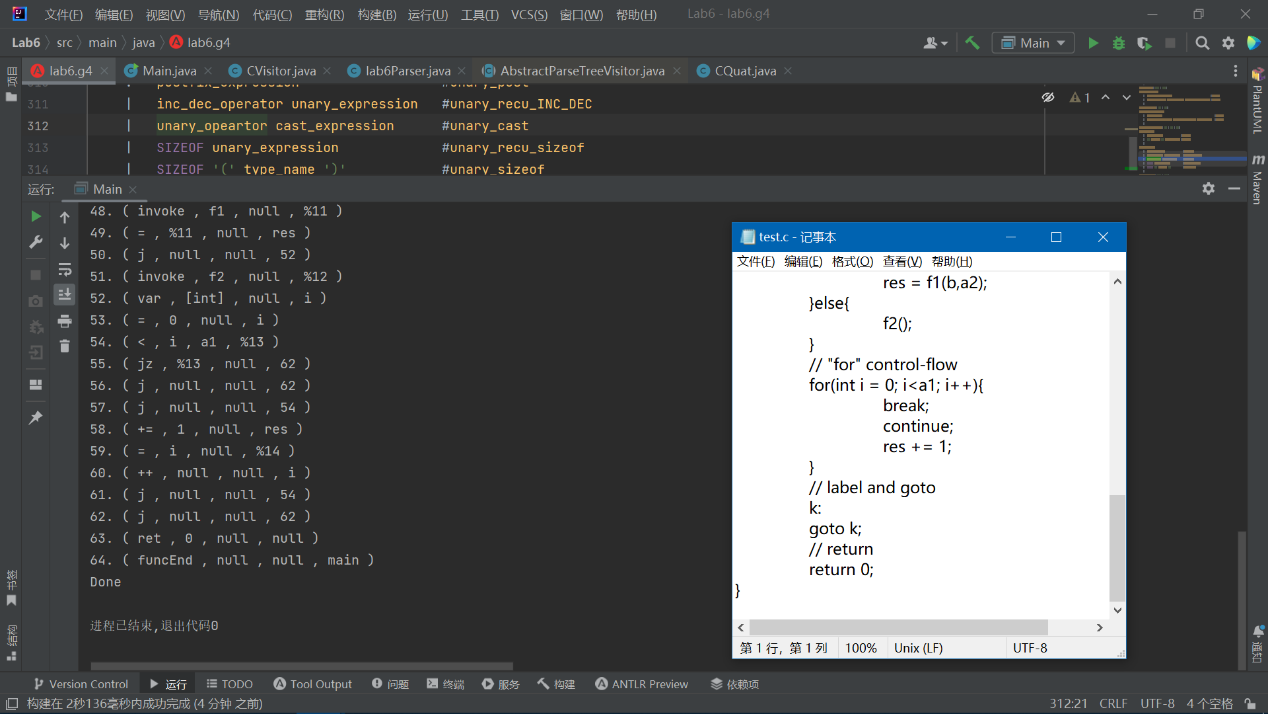
该语句是equality\_expression语句的递归备选分支，函数为单独的visit函数，其上下文Eq\_recuContext是Equality\_expressionContext的子类。

该函数是表达式visit函数的基本形式，因此以相等表达式的递归分支为例进行说明。

一般的表达式语法会分为直接定义和递归定义。直接定义的visit函数默认访问下一类表达式，并返回访问的值，因此不需要重写函数。visitEq\_recu则为递归定义分支的visit函数，首先分配一个内部使用的局部变量编号，以%开头。然后依次序访问左侧的equality\_expression、右侧的relational\_expression，因为equality\_operator为左结合性质。最后添加四元式，分别为相等运算符、左侧返回值、右侧返回值、局部变量。并以本次分配的局部变量为返回值返回。

1. 输出四元式列表，生成中间代码

在visitor遍历语法树完成后，所有的四元式都将按次序保存在CQuats数组中，遍历该数组，并按No. (op,arg1,arg2,res)的格式输出。

1. **运行效果截图**
2. example\_test.c
3. test.c
4. **实验心得体会**

本次实验是编译原理与设计的第七次实验。本次实验是在第六次实验的基础上进行的，大范围地修改了lab6的文法与visitor类。

在文法方面，我深入认识了文法递归定义的必要性与便捷性，递归定义使得后续visit的编写与访问更加便捷，体现了函数递归的本质。

在visitor遍历器中，我对几乎所有语法都编写了相应的visit函数，同时实现了语义检查和中间代码生成工作。优化了返回值的类型，改为String类型更有利于CQuat的编写，避免了Object强制类型转换造成的未知错误，代码更为规范。

增添了语义检查的内容，更全面的visitor编写使得检查语义错误变得更加清晰明了。对于中间代码生成，选择使用ArrayList数组来保存四元式列表，只需遍历一次语法树即可同时进行语义检查、中间代码生成的工作。最终只需遍历数组，按标准格式输出四元式即可。

本次实验让我第一次认识到了中间代码生成过程，又一次深入地阅读了C语言的规范文件。对于语法树的构成、visitor如何使用有了更多的心得体会。

本次实验我又一次修改了文法，基本重写了语义检查实验的内容，完整地完成了中间代码生成工作，实现了简单的控制流。收获颇丰，是一次非常有意义的实验。

但是由于递归调用，部分语句的构成略显冗余，如：

26. ( + , a1 , a2 , %4 )

27. ( = , %4 , null , res )

明显可以简化为：

26. ( + , a1 , a2 , res )

如此设置是为了兼容赋值语句中可能的‘+=’、‘\*=’等特殊赋值符号，但本程序中尚未解决上述冗余问题，不过简单的冗余可以在后续实验的优化步骤中进行修改，更精细的优化则需进一步分析。