**编译原理与设计**

**实验报告**

实验名称： Lab 8：目标代码生成实验

姓名/学号： 李昊阳/1120203053

1. **实验目的和内容**

**实验目的：**

（1）了解编译器指令生成和寄存器分配的基本算法；

（2）掌握目标代码生成的相关技术和方法，设计并实现针对x86/MIPS/RISC-V/ARM的目标代码生成模块；

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

**实验内容：**

基于BIT-MiniCC构建目标代码生成模块，该模块能够基于中间代码选择合适的目标指令，进行寄存器分配，并生成相应平台汇编代码。

如果生成的是MIPS或者RISC-V汇编，则要求汇编代码能够在BIT-MiniCC集成的MIPS或者RISC-V模拟器中运行。需要注意的是，config.xml的最后一个阶段“ncgen”的“skip”属性配置为“false”，"target"属性设置为“mips”、“x86”或者“riscv”中的一个。

如果生成的是X86汇编，则要求使用X86汇编器生成exe文件并运行。

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 22H2

Java: jdk-17

Assemble: MASM32 SDK version 11

IDE: IntelliJ IDEA 2022.2.5 (Community Edition)

Microsoft Visual Studio Community 2019 16.11.26

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，基于上一次中间代码生成的基础，完成目标代码生成模块。虽然指令选择和寄存器分配都存在多种算法，但是本实验最重要的目标是能够生成目标代码，并在相应的系统平台（x86）或者模拟器（MIPS，RISC-V）中运行，走完编译器设计的整个流程。386 汇编需要配置好 masm32。

在前置实验lab1-7中，我都是在Windows平台下完成实验的，且本人更为熟悉80386汇编语言（在上学期的汇编语言课程中，我分别学习了Windows与Linux平台的汇编语言基础），因此选择在lab7的基础上，继续在Windows平台上完成lab8的所有步骤，目标语言为80386汇编语言。

为了进一步生成切实可行的目标语言，仅靠翻译四元式是完全不够的，还需要语义检查、中间代码生成模块的密切配合，因此先对之前的lab7所作工作做进一步改进，接着再编写编译模块。

以下为优化lab7的步骤。我在lab7的CVisitor基础上进行了修改，复制为CInterpreter类，以有别于旧代码。

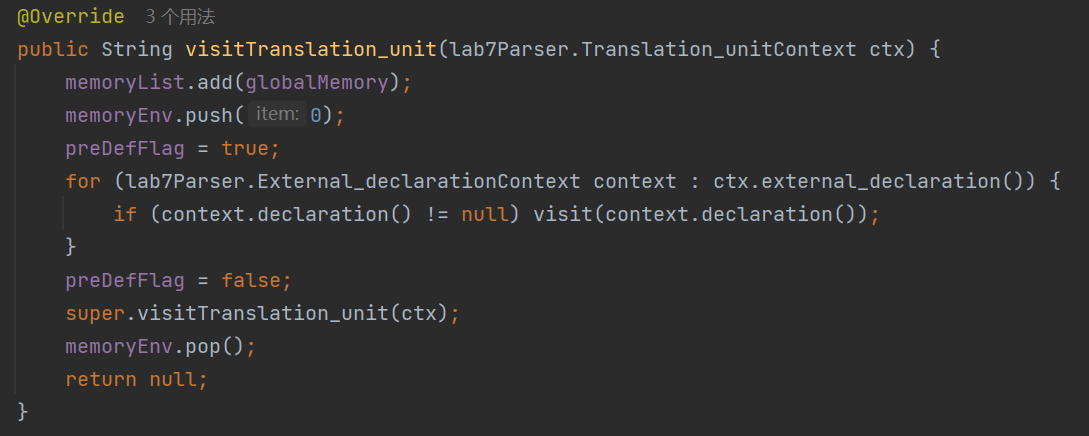
1. 重组织四元式生成顺序，提前声明变量

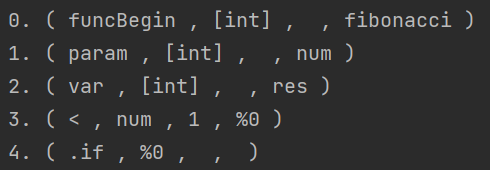
根据汇编语言的性质，变量可储存在data段或bss段、堆栈区、寄存器等地方，而要将声明的变量正确存取，首先得考虑变量的存放位置。对于全局变量（即在函数外定义的变量），应在data段进行定义并初始化赋值。对于局部变量，应在函数中定义为局部变量，并存放在栈中固定偏移量处。对于临时变量，则根据情况，灵活储存在寄存器、栈、内存中。

因此，在程序开始时定义所有全局变量，在函数开始时定义所有局部变量，更有利于变量的调用，也更便于汇编代码的生成。

首先定义所有变量，在经过深思熟虑后，发现更改语法树的访问次序、先遍历所有声明语句是最简单有效的方法。因为部分变量会在定义时赋值，打乱了后续的变量定义（若有），有悖于我们先定义再使用的思想，因此设置

private boolean preDefFlag;

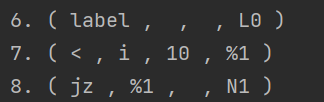
预定义标志位，true表示处于提前声明阶段，false表示正常访问。如将要访问external\_declaration结点，此结点可能是全局变量定义，也可能是函数声明。首先置preDefFlag，访问所有是全局变量定义的结点，重置标志后，再正常访问所有结点，以确保先定义所有变量，再进行初始化、函数定义的操作。

 在复合语句中也是同样处理（仅在translation\_unit与compound\_ statement的子结点会出现声明语句），使得函数起始呈现函数定义、参数定义、局部变量定义、初始化赋值、语句等的结构。

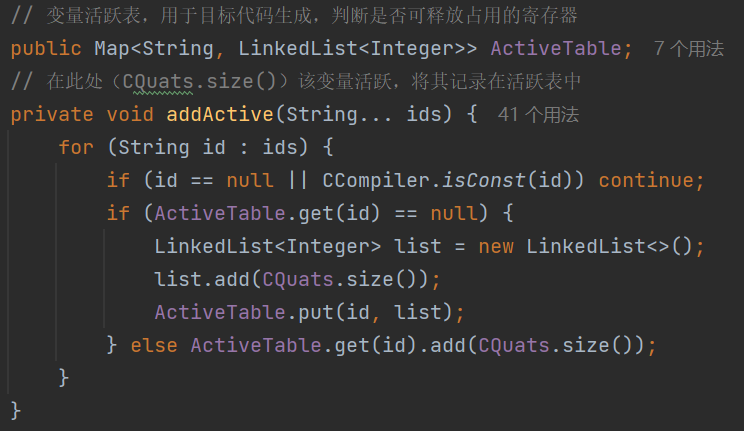
为了更好地配合preDefFlag，对visitDeclaration等visit函数也作了修改。若标志为真，则忽略初始化语句，仅进行定义；若为假，则访问所有赋值语句，并生成相应四元式，不再定义。

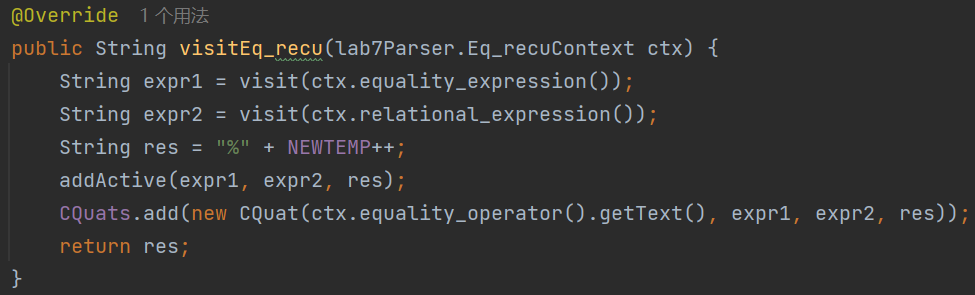
1. 优化控制流，增加活跃表

对于四元式的生成，优化了控制流。增加了.if伪指令四元式，使得选择语句的结构更加清晰。但对于while、for等循环，依旧使用最基本的标签与跳转语句，完整的体现了控制流的过程，也解决了多种多样的循环条件难以翻译成.while或.repeat伪指令的问题。

且在lab7中，省略了标签语句，将跳转行数直接写入四元式中，这其实是非常不便的，不利于目标代码的重组织。因此增加了label四元式，将jmp、jz等跳转目标设为标签，标签名自动生成、以L或N开头，L代指Label、N代指Next，并分别在循环语句的开始与结束处生成，或由跳转语句指定生成。

在编译成汇编语言的过程中，需要频繁地判定一个临时变量是否会被再次使用，以此来决定是否释放它所占用的寄存器。因为寄存器是系统中的稀缺资源，编程者可见的常用寄存器仅有eax、ebx、ecx、edx、esi与edi，以及指针寄存器esp和edp。我们无法永远保存所有的临时变量，不再使用的局部变量的值则应该被及时覆盖使用，以大幅提高计算机资源的利用率。

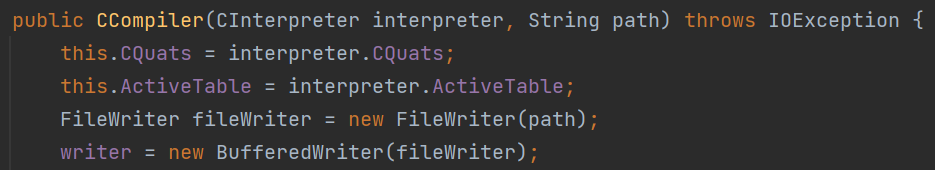
在生成中间代码的同时生成活跃表ActiveTable，在活跃表中记录所有变量的调用信息，并以键值对(key, LinkedList<Integer>)保存，链表中每个值为调用该变量四元式的序号。活跃表与添加函数如下：

如在相等表达式中，arg1、arg2、res的变量都将记录一遍：

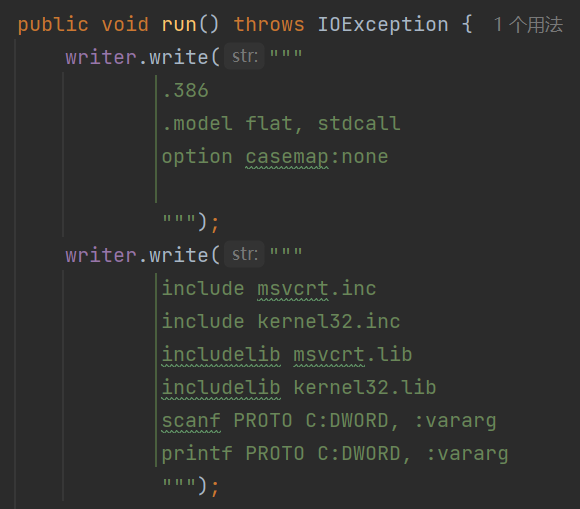
根据活跃表中对应变量的链表信息，最后一个值为该变量最后一次调用的序号。若当前访问序号大于该序号，则可以轻松释放它占用的寄存器，并覆盖它的值。若相等，则需具体情况具体判断。

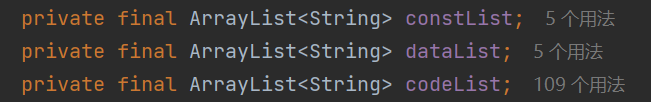
以下为生成目标语言的步骤。新建CCompiler类，参数为CInterpreter实例，并由Main函数调用，最终生成指定路径的.asm汇编文件。

1. 新建CCompiler类，编写run()方法

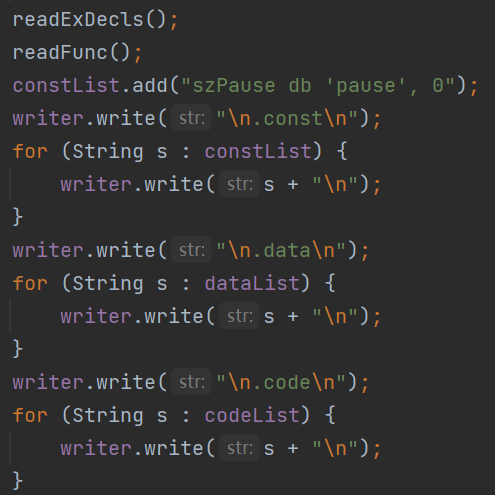
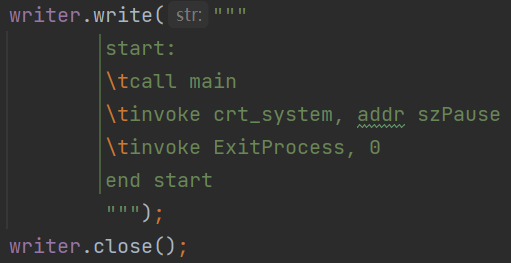
在CCompiler类中，首先初始化所有成员，指定将要使用的四元式列表与活跃表，并实例化BufferedWriter，将生成的汇编代码写入指定文件路径中。

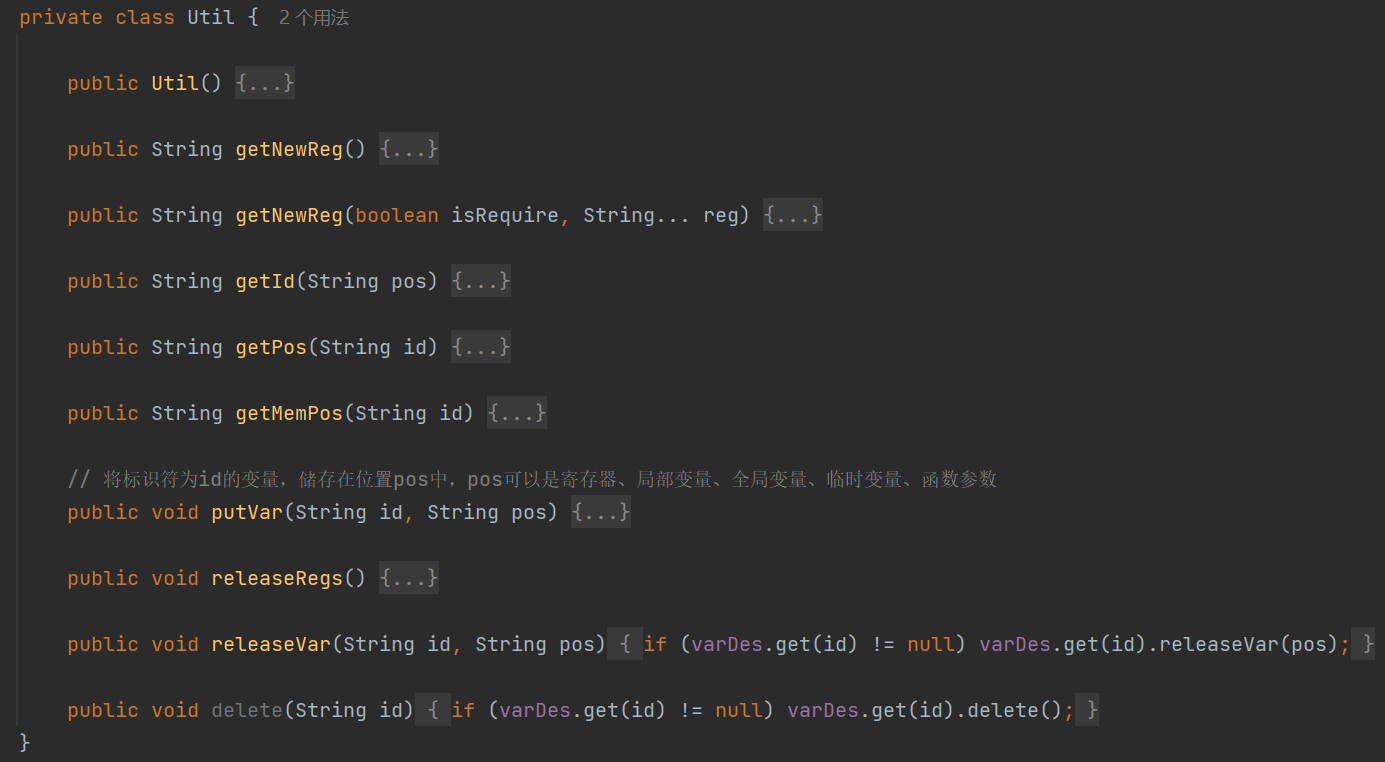
而后由Main函数调用run()方法。run()是编写目标代码的主要方法，按顺序生成伪指令配置，添加库函数依赖，定义常量段.const与数据段.data，然后根据四元式生成目标汇编代码段.code，最后生成start函数。

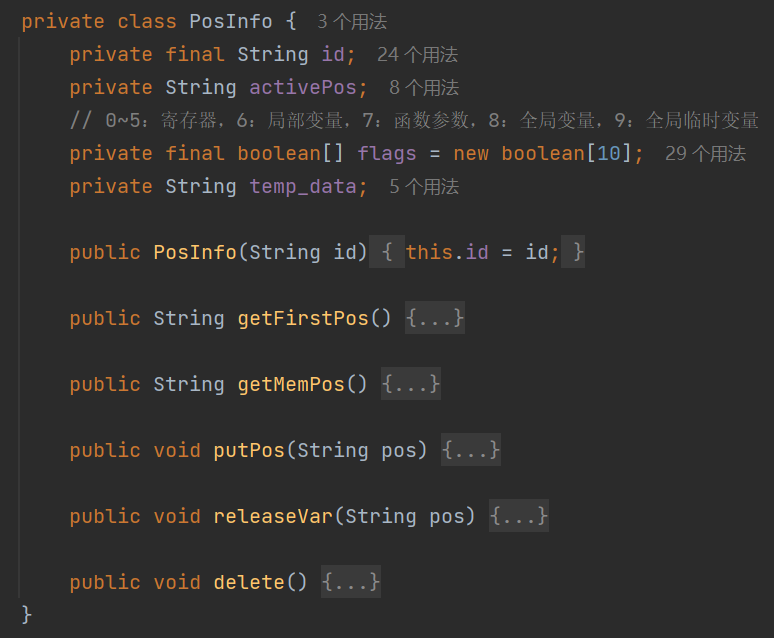
 本次实验选用80386汇编语言，指定内存模式为平坦模式flat，参数传递时使用标准调用stdcall，并指明大小写敏感；对于测试用例中的Mars\_PrintInt、Mars\_PrintStr、Mars\_GetInt等函数，改写为printf与scanf库函数，因此需添加多个库依赖，并声明printf与scanf库函数：

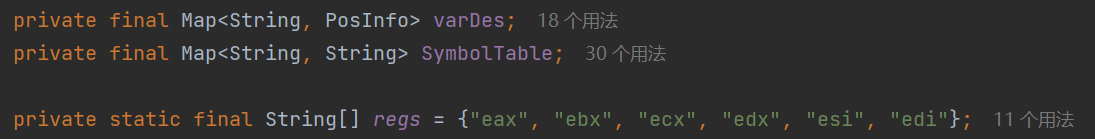
 由于生成目标代码时需要重组织，将数值/字符串常量等写在.const段，将全局变量写在.data段，其余代码写在.code段，因此需设置三个临时列表。先遍历扫描四元式数组，将生成的汇编代码储存在对应列表中，最终依次写入不同段中。

readExDecls()与readFunc()遍历四元式列表，分别对应语法成分中的外部声明与函数。

 最后编写start部分，用来调用main函数，最后调用暂停函数system (“pause”)以避免exe程序运行结束后直接退出。

1. 编写Util、PosInfo等内部类

Util为工具类，主要提供的方法有：分配新寄存器、获取对应地址的变量名、获取变量名所在的地址、将变量的值储存在指定地址中、以及释放寄存器，此处地址泛指寄存器、局部变量或全局变量。

PosInfo为地址信息类，每个对象记录在运行时一个变量当前值的存储位置（被分配位置），可以是寄存器、局部变量、全局变量等，并指明活跃处（最新值）。所有变量的地址信息储存在varDes地址描述符中，类型为Map<String, PosInfo>，String值为变量标识符。

SymbolTable为符号表，记录所有可使用地址的内容，包括6个可用寄存器，若未使用则为null，类型为Map<String, String>。

1. 编写readExDecls()方法

在外部声明的四元式中，以op分类，有‘var’、‘array’、‘pointer’类型全局变量声明，‘=’对全局变量初始化，‘funcDef’函数声明。前三类调用globalDeclaration()方法，根据类型限定符生成相应格式的汇编代码，并添加至dataList中。所有未初始化的全局变量均置‘？’，若通过‘=’进行初始化，则调用adjustInit()方法，搜索dataList中对应项，并修改初始化值。函数声明则使用proto伪指令。

 其中，ascertainType()方法生成对应类型描述符的汇编格式：

1. 编写readFunc()方法

该方法是生成目标代码的核心方法，在全局设置一个index指针，循环读取四元式列表，每扫描到一个‘funcBegin’函数定义，则开始循环读取函数内部四元式，生成相应的函数主体，直到读取到对应的‘funcEnd’，开始扫描下一个函数定义，或列表结束出错。

需注意的是，寄存器分为调用者保存寄存器与被调用者保存寄存器，以此来保证函数调用时不会更改寄存器中的有效值。按约定，ebx、esi、edi为被调用者保存寄存器，因此在函数定义时统一使用uses伪指令，在函数被调用后保存寄存器的值到栈中，无论其是否需要被保存。

函数定义使用proc与endp伪指令，调用readPara()读取该函数的参数四元式‘param’，并以如num:dword的格式添加至proc伪指令右侧，这些参数指明了长度，将以stdcall方式进行传参。

随后调用readLocalDeclaration()读取该函数的所有局部变量定义，并使用local伪指令表示，如local res:dword。这些局部变量顺序保存在函数的栈帧中，并以[esp-offset]的格式访问，在编写汇编代码时，可直接使用标识符res。

根据四元式op的不同，处理方式也不同。下面介绍典型的几类：

* 1. arg、invoke

arg与invoke功能是调用函数并传参。对于是字符串常量的参数，自动生成标识符并添加至constList中。生成参数列表后、执行invoke前，查询调用者保存寄存器eax、ebx、edx是否正在使用，若有，则保存至栈中，待函数调用返回后再复原。

因为函数返回值默认保存在eax中，因此对于保存在eax中的值有如下处理过程：

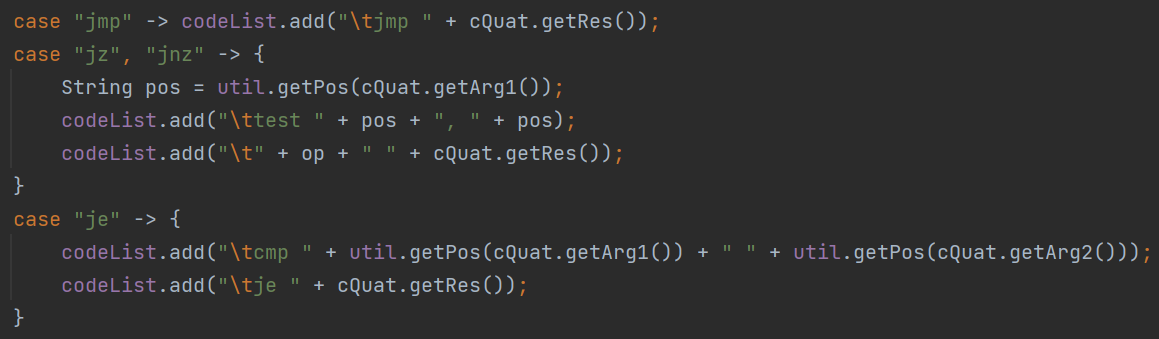
直接使用以下语句：util.putVar(cQuat.getRes(), "eax");

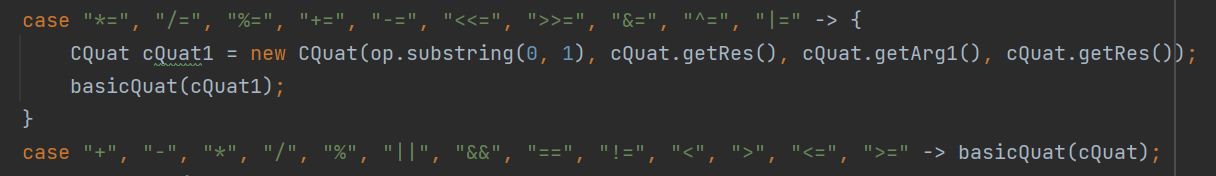
util类的putVar()方法调用对应PosInfo类的putPos ()方法，修改PosInfo的描述信息，调整活跃位置，并写入SymbolTable中。SymbolTable写入后返回目标位置中是否有旧值，若有，则释放对应变量在当前位置的值，若释放的是变量活跃位置（储存有新值）且变量在查询ActiveTable后还将被使用，则将新值写回变量的其他储存位置，或选用一个新的寄存器保存。

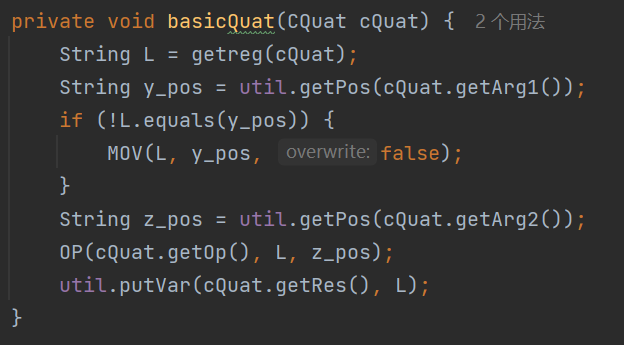
最终形成的函数调用语句如下：invoke fibonacci ,eax

* 1. jmp、jz、je

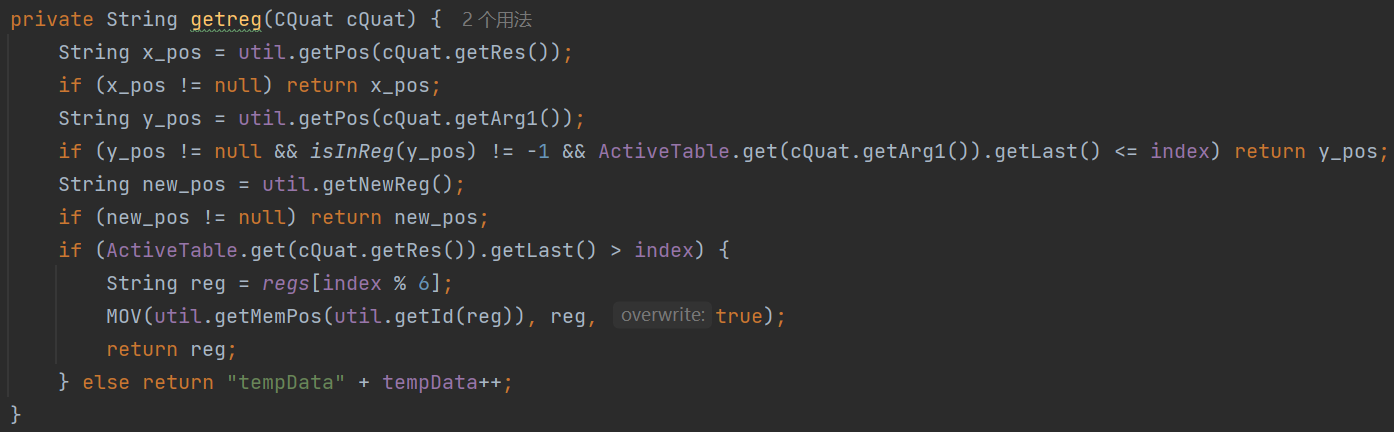
jmp语句直接跳转。条件跳转语句则需先测试或比较条件，jz使用test判零，je使用cmp判等，然后根据条件码跳转。

* 1. 二元运算，包括+、-、\*、/等四则运算，>、<等逻辑运算，\*=、+=等赋值运算

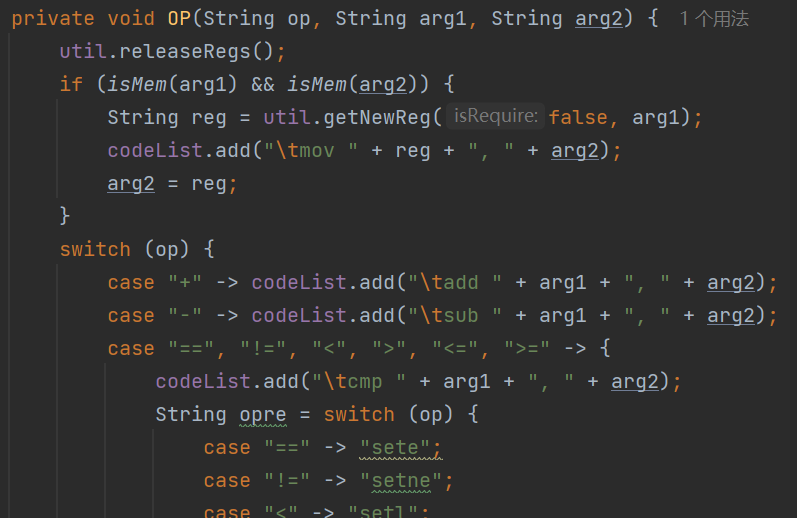
由于C语言中逻辑表达式的值真为非0、假为0，而非Java中特定的boolean类型，因此将逻辑运算与四则运算归为一类。赋值运算则拆分为先四则运算再赋值。编写basicQuat()、getreg()、OP()等方法来生成目标代码，并调用basicQuat(cQuat)。

basicQuat()：在方法中调用getreg(cQuat)方法，返回值为地址L，用于存放arg1与arg2的计算结果。若L与arg1的地址不相等，则将arg1的值移动到L中，即mov L, arg1。随后调用OP(cQuat)计算op L, arg2，生成目标代码。最后将地址L加入x的地址信息中，修改地址描述符。

getreg()：若res已分配寄存器或内存，则返回res的地址；若arg1在寄存器中，且arg1在执行后是不活跃的，则返回arg1的地址；若上述不能实现且存在空寄存器，则返回一个空寄存器；若res还将被引用或op是一个需要寄存器的运算符时，找一个被占用的寄存器，将其值移动到某一内存中（可以是局部变量、全局变量或全局临时变量），返回该寄存器；否则，返回一个全局临时变量。



OP()：对于任意双操作符运算而言，两个操作符不能同时为内存。因此需检测arg1与arg2的活跃位置是否都是内存，若是，则挑选任一空寄存器，将arg2移至该寄存器中，后续使用该寄存器代替arg2。

‘+’、‘-’、‘\*’对应汇编代码为add、sub、imul。‘/‘、’%‘对应汇编代码为idiv，需先将被除数移至eax中，cdq拓展后除以除数，商在eax中，余数在edx中。’&&‘、’||‘对应汇编代码为and、or。比较运算符’<‘、’>‘等对应汇编代码为setl、setg等，结果储存在指定8位寄存器中，再使用movzx指令拓展至32位寄存器。

* 1. 数组访问=[]与数组赋值[]=

数组访问采用多维指针的思想，每一维的访问返回的都是指向该维的指针。如对于二维数组int A[m][n]，第一次访问A[i]返回的即是int[n]类型的指针。使用Map<String, ArrayList<String>> quote保存每一次数组访问的信息，key值为res，可以是引用名%num或取地址名&num，value保存数组名与每一维的访问信息。每次进行数组访问时，先判断arg1是否以%开头，若无，则新建quote项，添加数组名与第一维信息；否则，访问quote中arg1指向的value值，将信息添加至新建quote项的前部，再添加本维度的信息。对于四元式：(=[],A,10,%1), (=[],%1,20,%2)，则quote中保存有(%1,[A,10]), (%2,[A,10,20])。

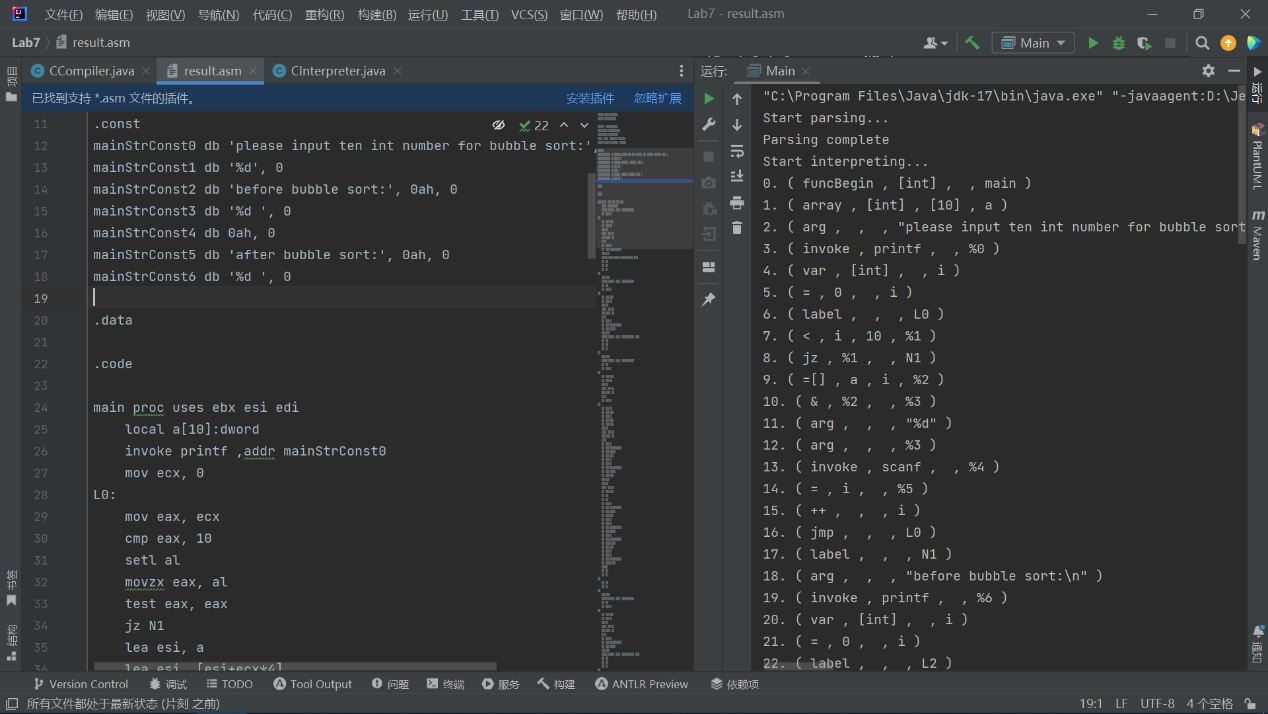
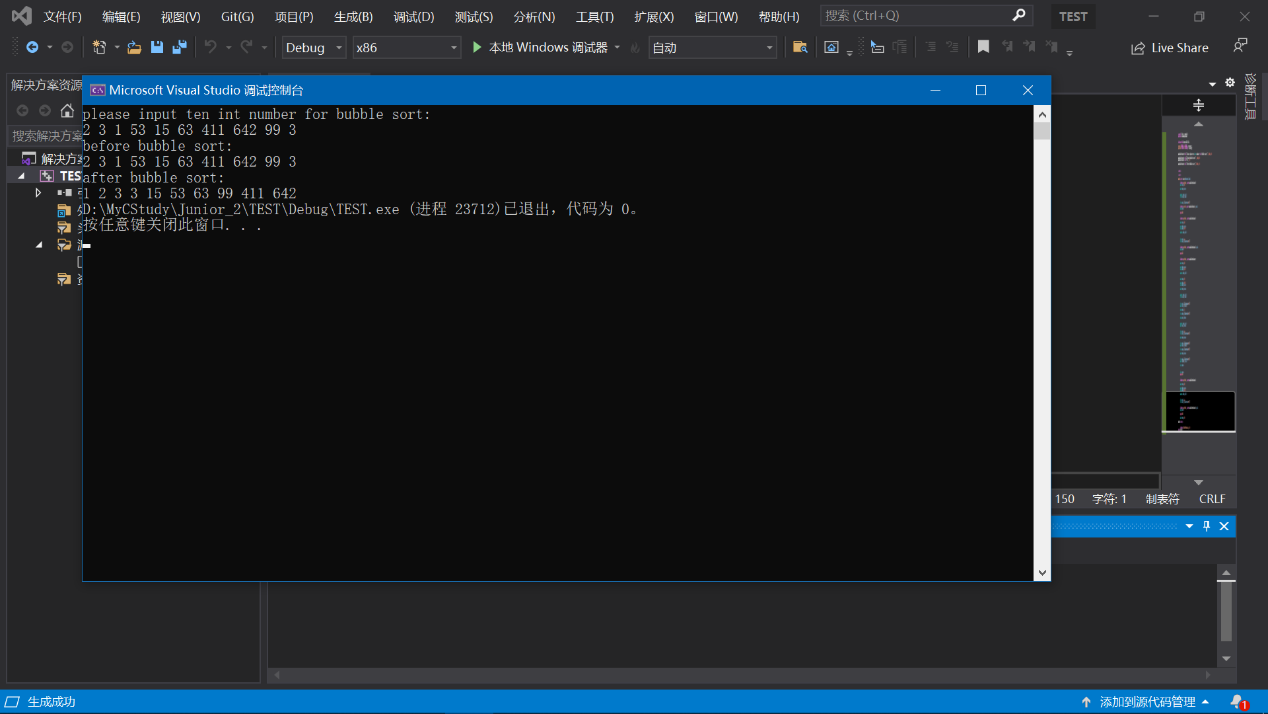
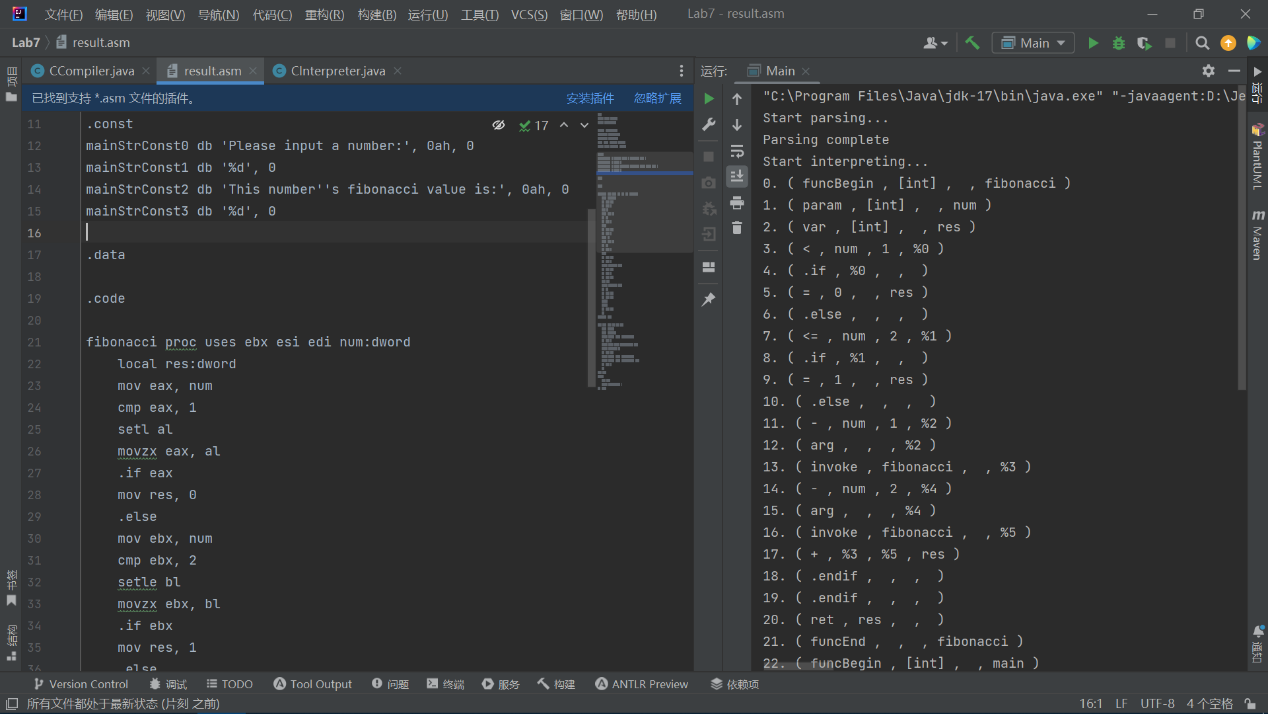
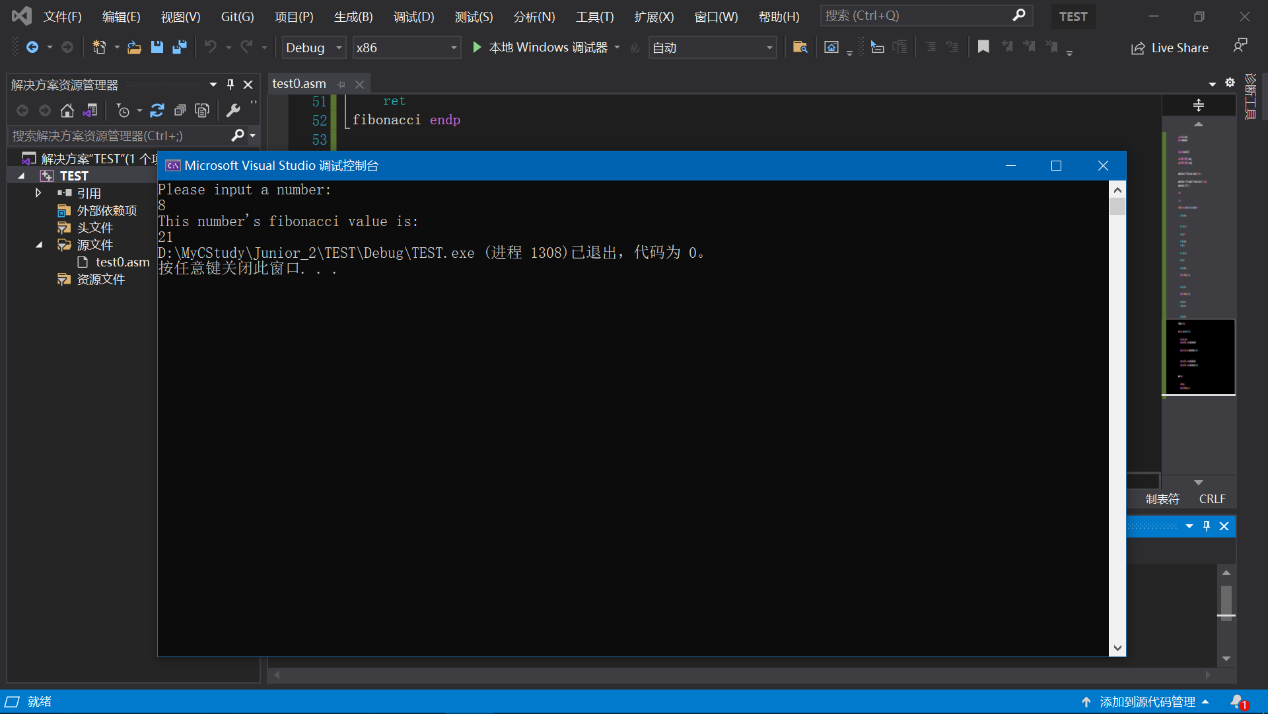
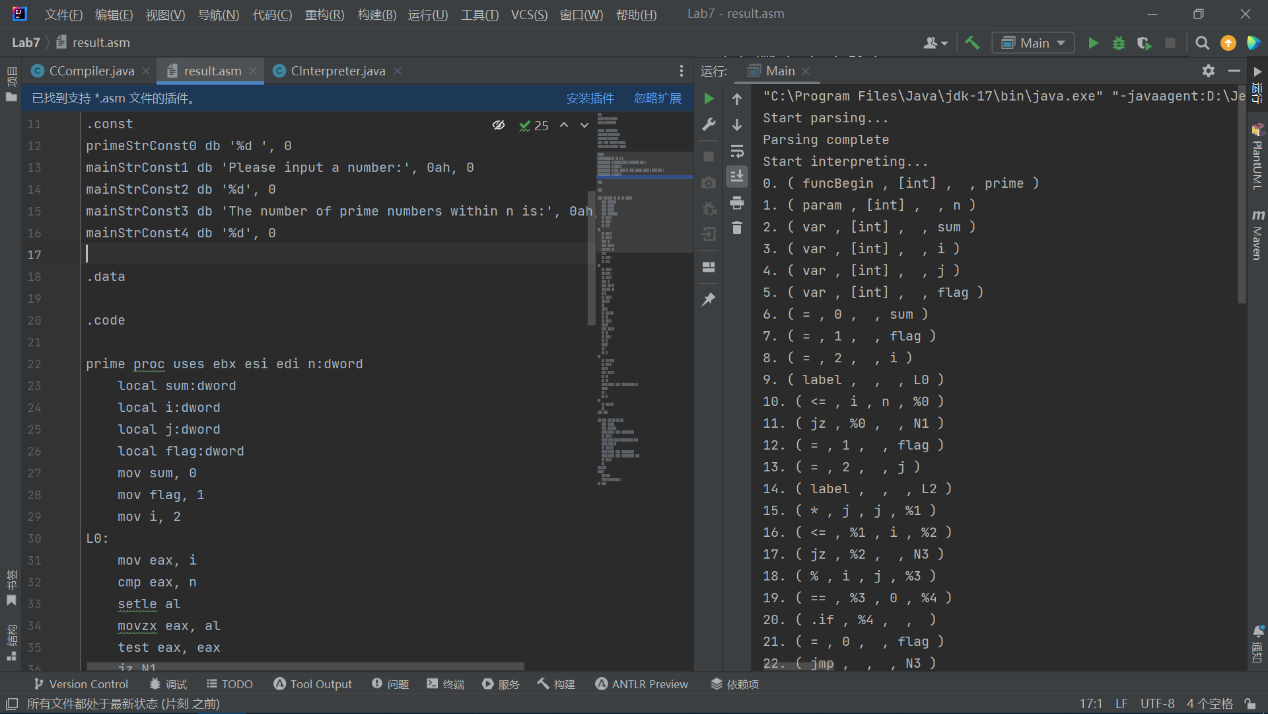
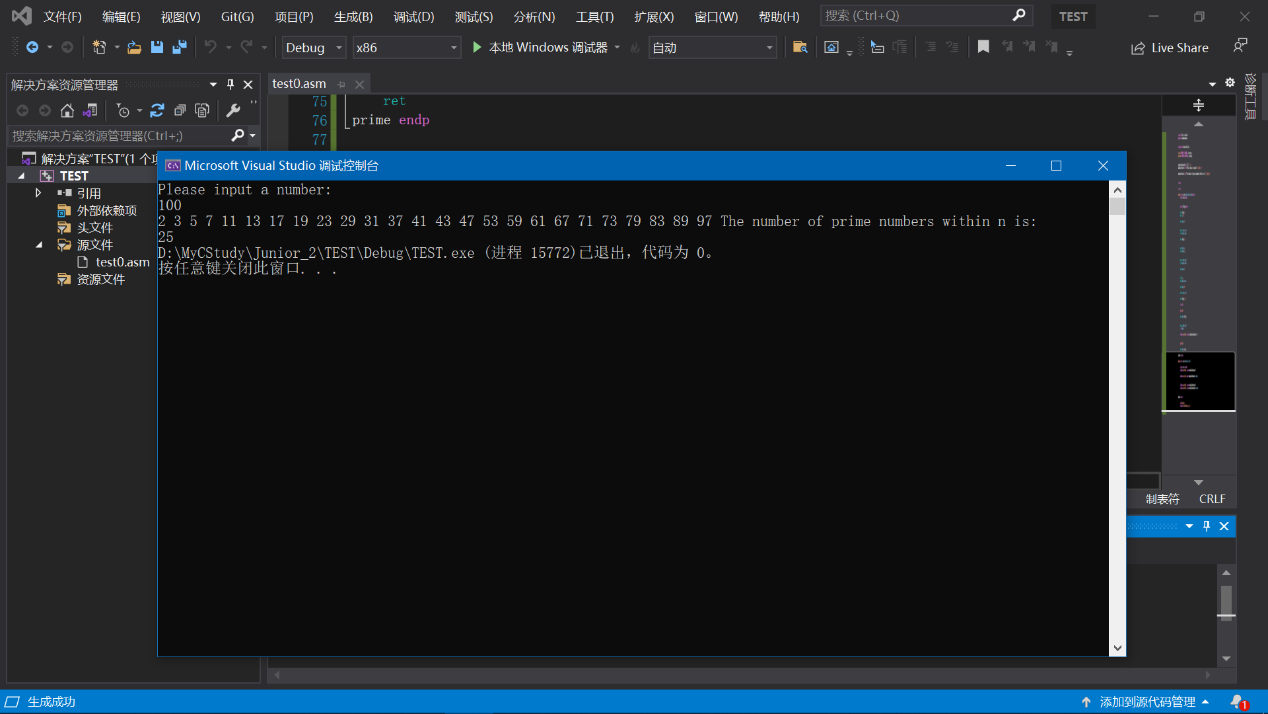
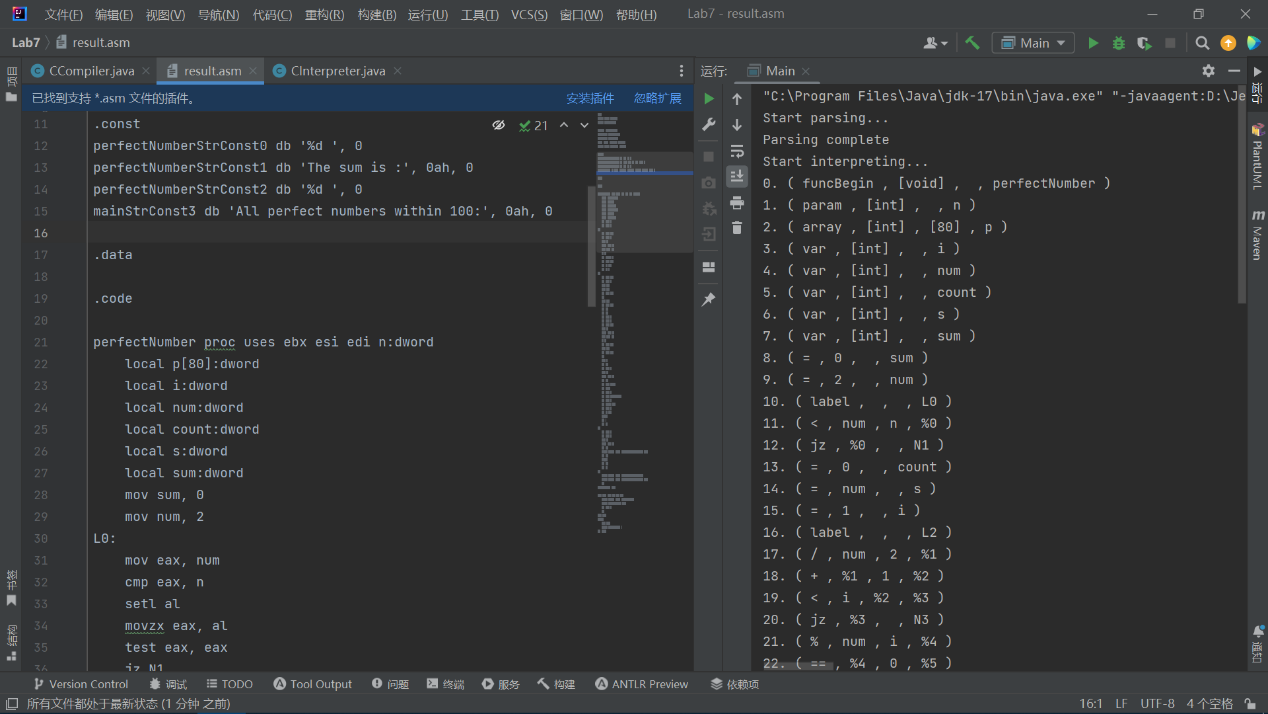
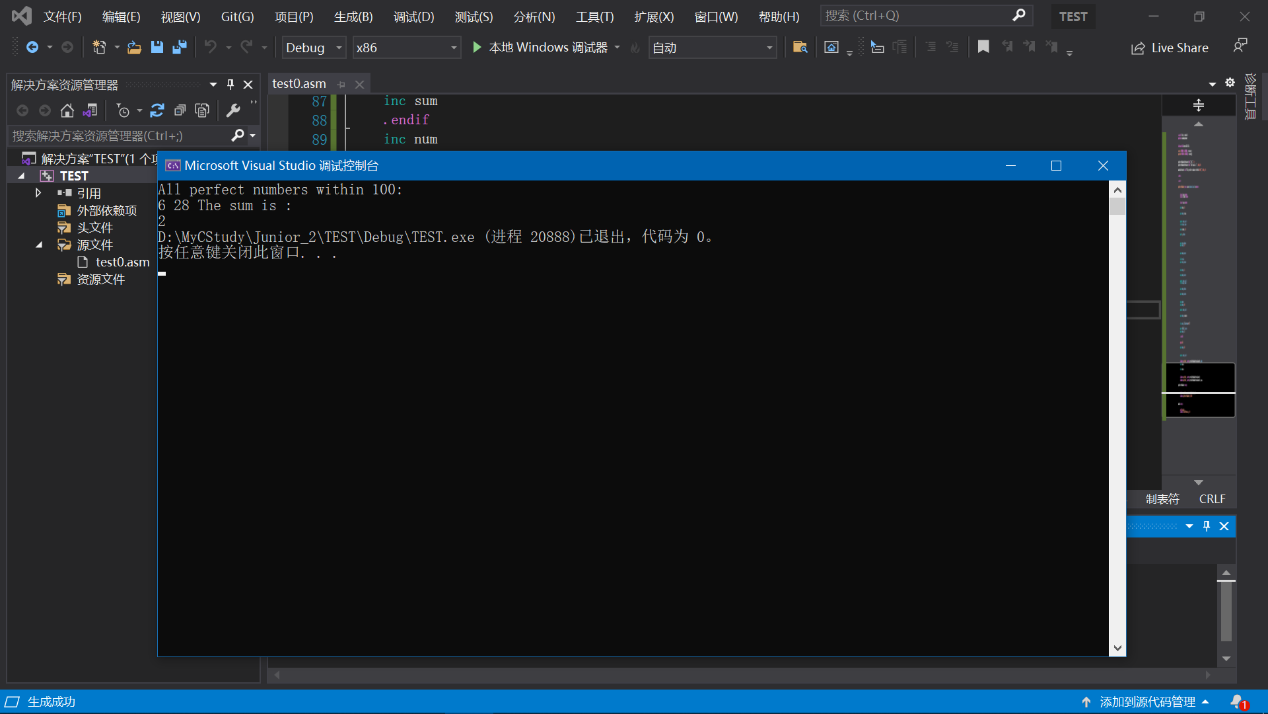
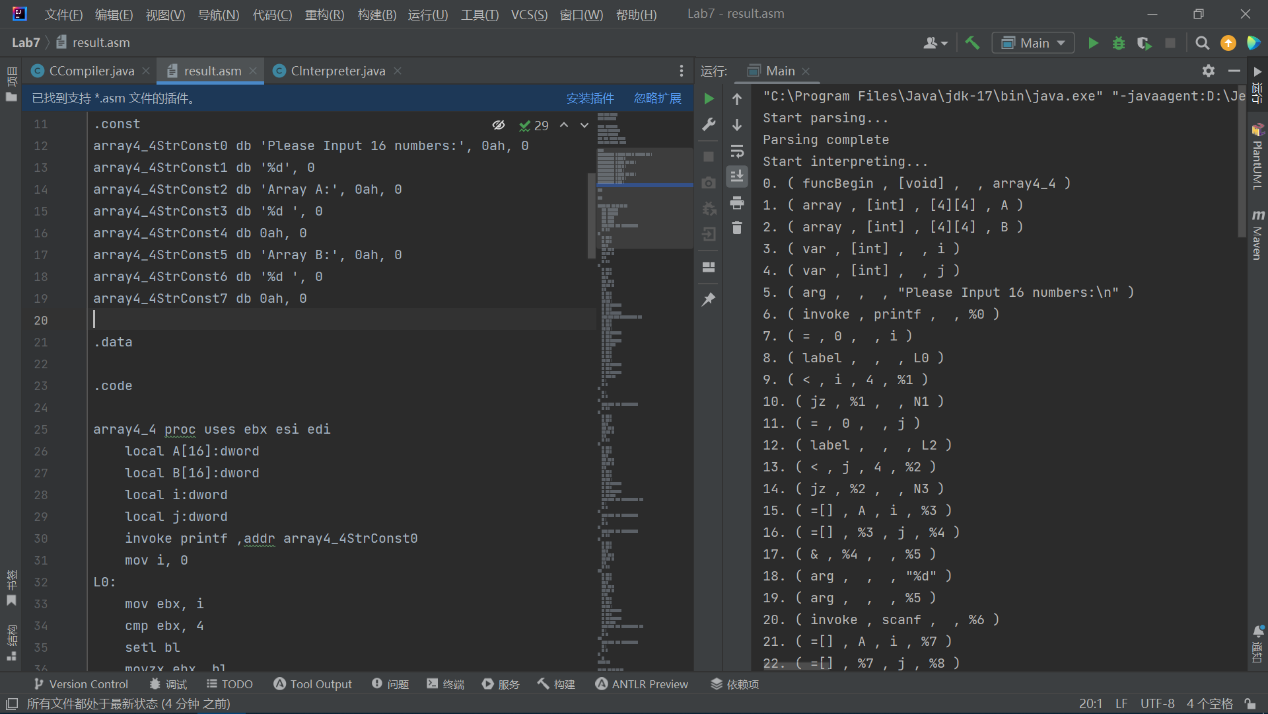
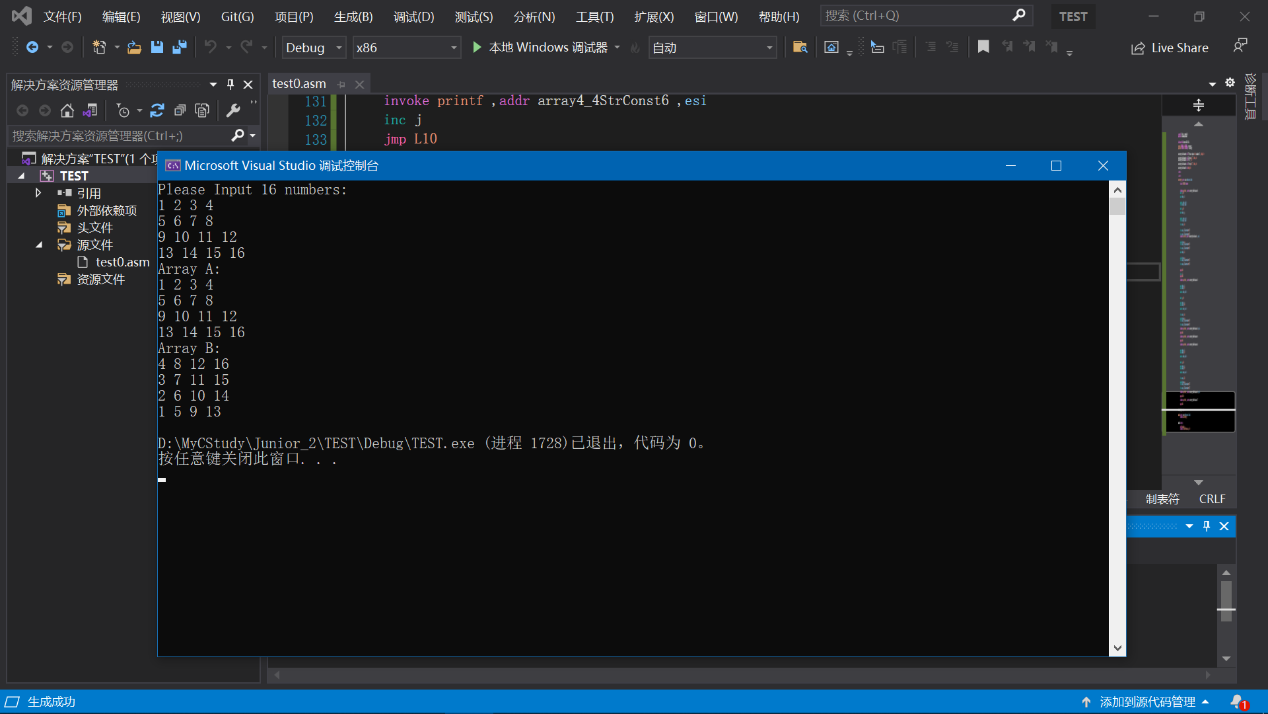
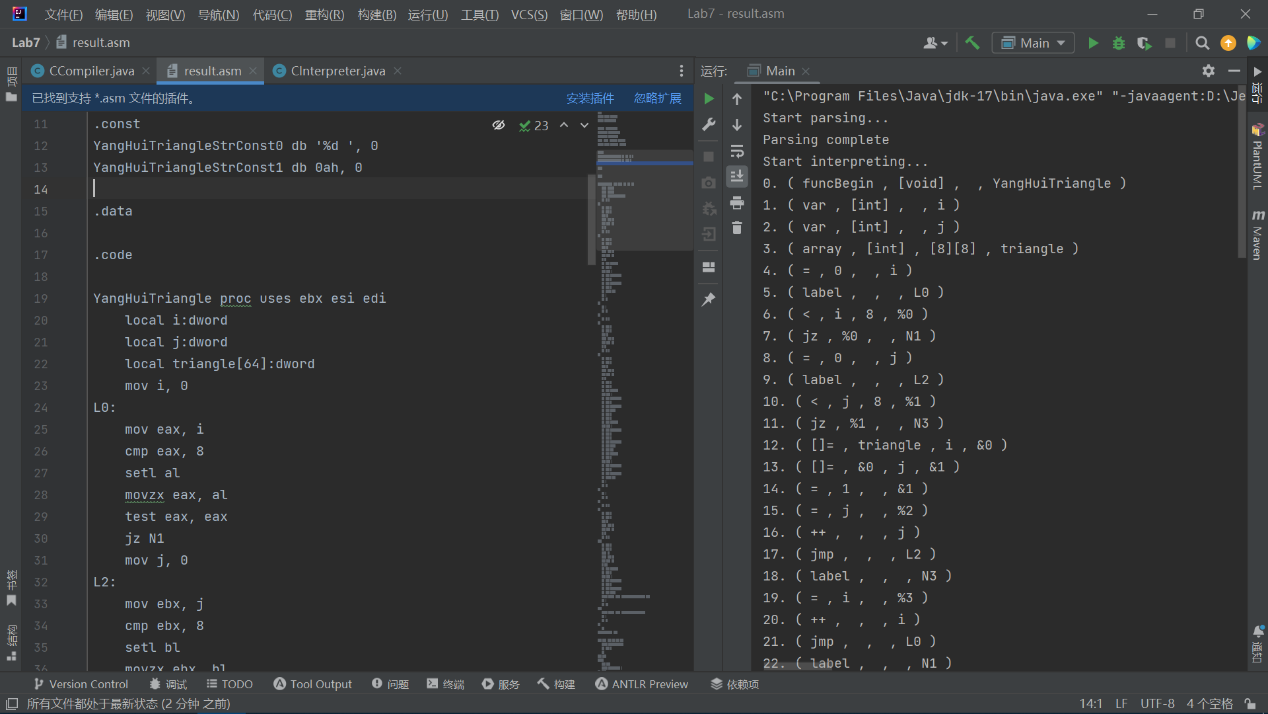
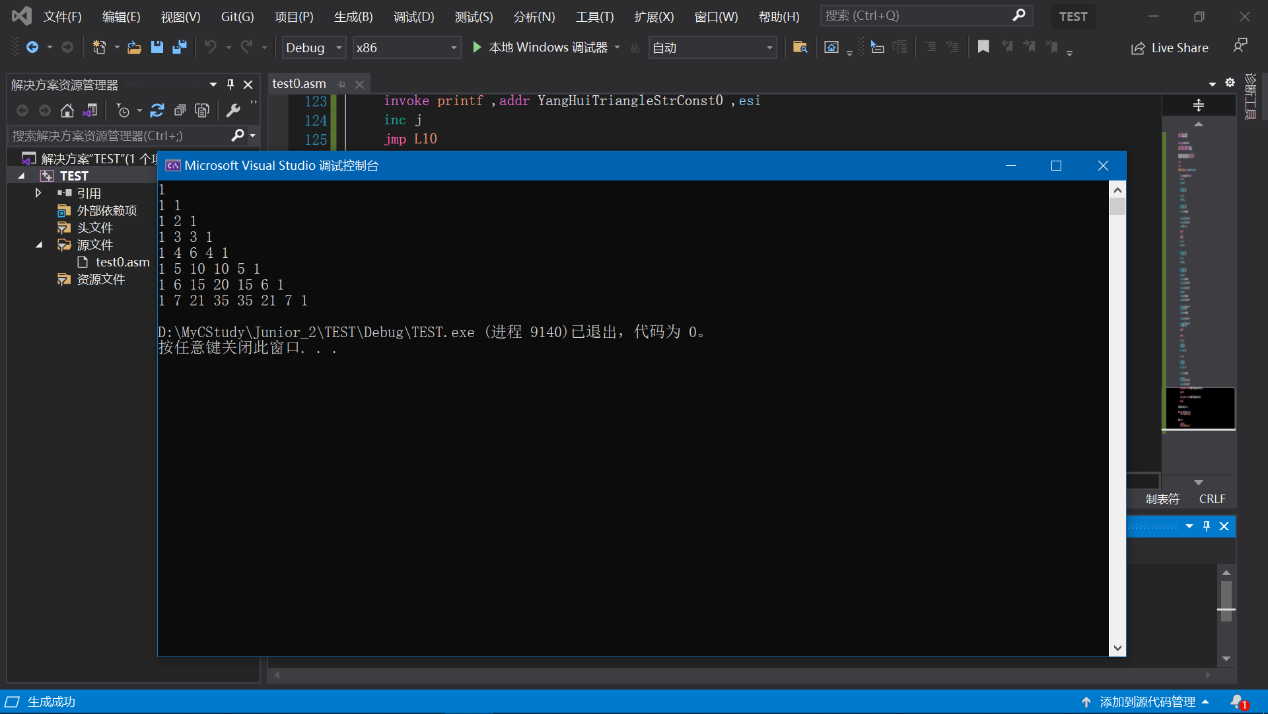
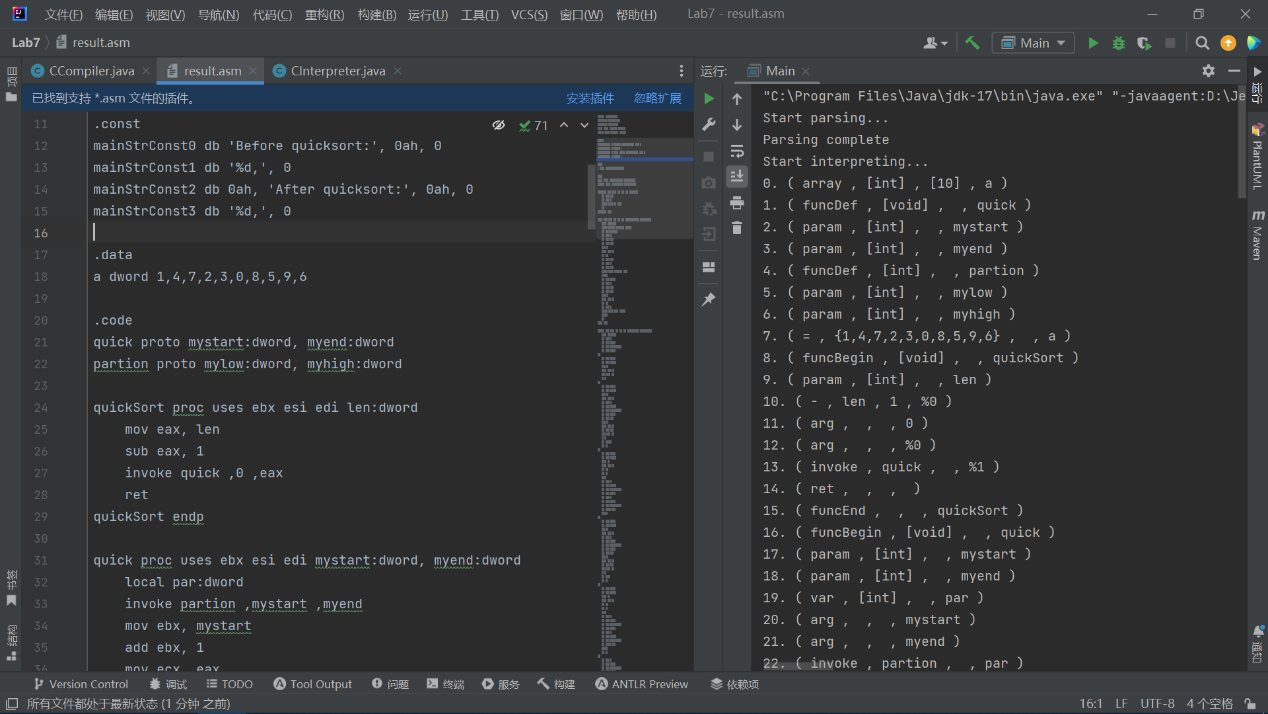
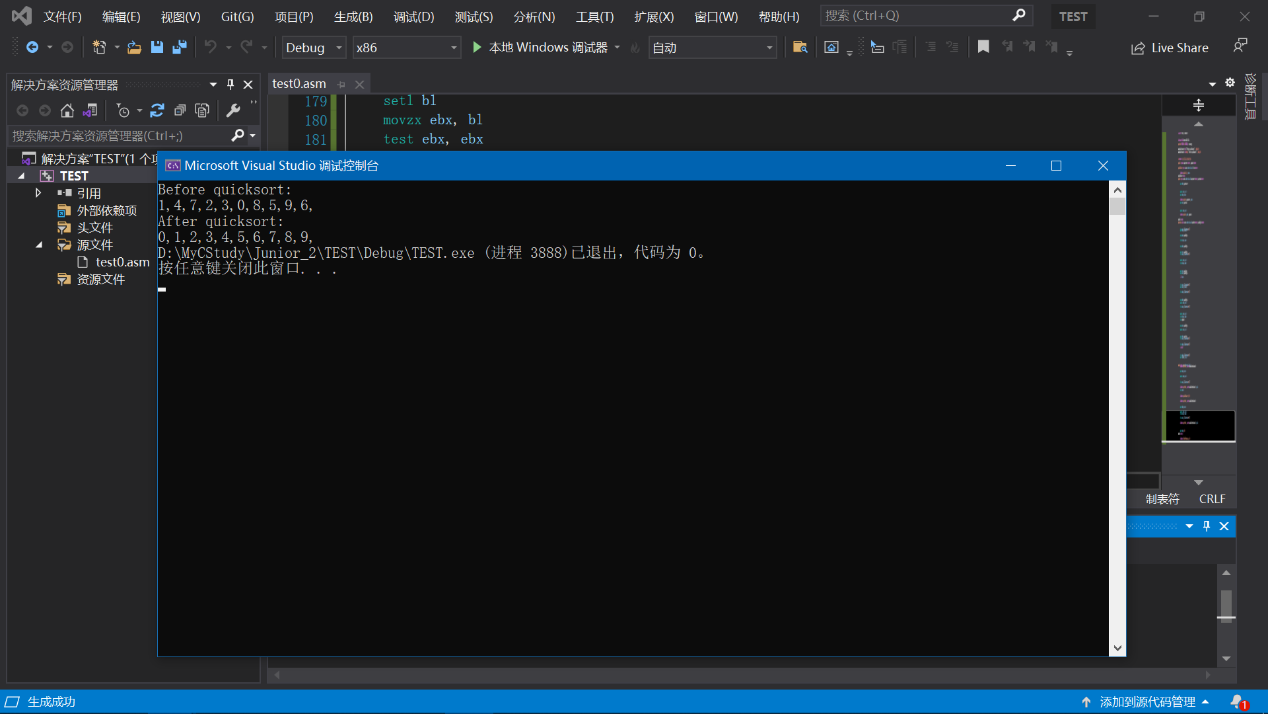
对于第一次的数组访问，申请一个新寄存器保存数组首地址，首选esi寄存器，然后将数组首地址移至该寄存器中lea reg, A。若arg2为常数，则直接使用lea指令，将指向该维的地址保存至reg中，先计算arg2\*size ，size为该维数组指针对应大小，该值保存在ArrayInfo中，然后生成指令lea reg, [reg+arg2\*size\*4]。若arg2不是常数，且保存在寄存器reg1中，则使用imul指令计算长度：imul reg1, size， lea reg, [reg+reg1\*4]。否则，需先将arg2保存至临时申请的寄存器reg1中：mov reg1, arg2，imul reg1, size， lea reg, [reg+reg1\*4]，其中reg1是不同于reg的寄存器。

进行以上操作后，需判断数组是否已经访问至最后一维。因为reg中保存的是对应的指针，若需使用其值，还需将值移动至reg中：mov reg, [reg]。

数组赋值操作与上述类似，需要注意的是最后赋值阶段的四元式op为’=’，因此数组赋值操作应在’=’中进行。且最后应将res值移动至reg指向的位置中：mov [reg], reg1，其中reg1保存的是res的值。

1. 最终生成

最后进行系统检查，修改后对每一个例子生成asm代码文件与exe可执行程序。运行时，命令行参数指定为源程序路径。最终除7\_Dijkstra.c外，完成了7个测试用例的生成与运行。

1. **运行效果截图**
2. 0\_BubbleSort.c
3. 1\_Fibonacci.c
4. 2\_Prime.c
5. 3\_PerfectNumber.c
6. 4\_CounterClockwiseRotationArray4\_4.c
7. 5\_YangHuiTriangle.c
8. 6\_QuickSort.c
9. 7\_Dijkstra.c

本测试用例生成时仍有报错，由于时间与精力问题，该测试用例不再实现。

1. **实验心得体会**

本次实验是编译原理与设计的第八次实验，也是最后一次实验。本次实验是在第七次实验的基础上进行的，在一定范围内重写了lab7的visitor类，并编写了CComplier类。

在visitor遍历器中，我对中间代码生成工作再次进行了优化。增加了多项四元式类型，修改了变量声明次序、控制流表达。增加了变量活跃表等功能，以更好地实现目标代码生成。

目标代码生成是基于80386的汇编语言。我在以往的汇编语言中多次使用该语言，并编写了汇编游戏，因此颇为熟悉。参照教材上的多个算法，我对四元式依次进行了翻译重构，在多次测试与修改后，得到了想要的结果。

本次实验是第一次从源代码到目标代码的完整编译生成过程，是集编译原理这门课程的大成之作，饱含了以往所有实验的心血，整个实验周期极长，收获颇丰，获得了极好的结果。

在本次试验中，我对四元式优化部分作了很多工作，删去了冗余的四元式和未曾使用的赋值等语句，解决了第七次实验提到的问题。但实际上，我并未实现编译过程中的优化步骤，未曾实现任何对目标代码的优化，如循环优化、基本块优化等。优化同样也是编译原理中重要的一部分，我已在教材中进行了深入的学习。但是真正实现优化算法更比目标代码生成困难，因此本实验只实现了目标代码生成。待日后行有余力时，再来补足不迟。