说明：之前提交过一次，但是存在着生成冗余代码的缺陷，因此和助教沟通后，我根据在编译原理课程中学习的内容重新修改了寄存器分配算法，请以此次为准进行给分。谢谢老师和助教老师！

LAB4 Spiglet_To_Kanga

这一节的任务是将 spiglet 代码翻译成 kanga 代码，二者最大的区别是 kanga 语言的寄存器是有限的，因此最重要的任务就是寄存器分配和栈的使用。

按照老师在课上的提示，我将任务分解成了四个部分：基本块划分、活性分析、寄存器分配、翻译生成。

如此分解的好处是，各个部分相对独立，耦合性低，比较方便调整具体的算法实现。这四个部分所共同使用的数据结构，是对 spiglet 中的语法成分的描述，包括基类 MySpigletType、子类 MyGoal、MyProcedure、MyStmt、MyTemp 等。下表总结了上述各类的属性和方法。

类名	描述	属性	方法
MySpigletType	基类，作为 visit 接口传递的参数类型	无	无
MyGoal	子类，相当于符号表	HashMap<String,MyProcedure> table	getProcess
MyProcedure	子类，相当于过程的符号表	HashMap<Integer, MyTemp> temps:过程出现的 temp 变量 Vector<MyStmt> stmtlist:语句列表	addstmt addlabel

		int paranum,maxparanum:参数个数,调用过程的最多参数 Int spillnum, tempspillnum:溢出单元个数，参数和寄存器分配时溢出的单元个数 Vector<Integer> blockloc: 记录块头指令位置的列表 HashMap<String, Integer> labels: 标签和位置的哈希表	
MyStmt	子类，记录一条语句的各种属性	int index: 位置 HashMap<Integer, Integer> livevars: 本句的活跃变量 boolean blockend: 是否块尾 boolean labeled = false: 是否有标签 String label: 标签	Addlive Deletelive Addnextlive Addjumplive liveschanged
MyTemp	子类，描述一个 temp 的寄存器分配情况	Integer index=0: temp 变量的序号 int reg = -1: 寄存器序号 int spill = -1: 溢出单元序号 boolean spilled=false: 是否溢出 HashSet<Integer> interferences: 干扰变量集合	无

接下来我分别介绍四个部分的实现。

4.1 基本块划分

基本块指的是程序流图中的基本结点，也就是只能从块中的第一

个指令进入、从最后一个指令离开的连续指令序列。进行基本块划分的目的是方便活性分析，因为如果是基本块内部的指令，则其活性只由其下一条指令和它自身内容决定，基本块的最后一条指令则由基本块的后继结点和它滋镇内容决定。

基本块划分的实现很简单，只需要一趟遍历 spiglet 的 AST 即可。对于一个过程，要顺序地访问语句：

(1) 如果是第一条指令，则判定为块头

(2) 如果是 CJUMP 或者 JUMP 语句，则判定为当前块的块尾，下一句为块头。对于目标 label，若在已访问 label 表中，则直接设为块头；否则放入到待访问 label 表中。

(3) 如果当前语句前有 Label，若 label 在待访问 label 表中，则判定为块头；否则压入到已访问 label 中

(4) 最后一条指令，判定为块尾

表示块头与块尾的数据结构也非常简单，对于每一个过程对象，会有 Vector<Integer> blockloc 来记录每一个块头指令的索引；对于每一个语句对象，会有 boolean blockend 来记录是否是块尾。

4.2 活性分析

在基本块划分的基础上，进行活性分析。

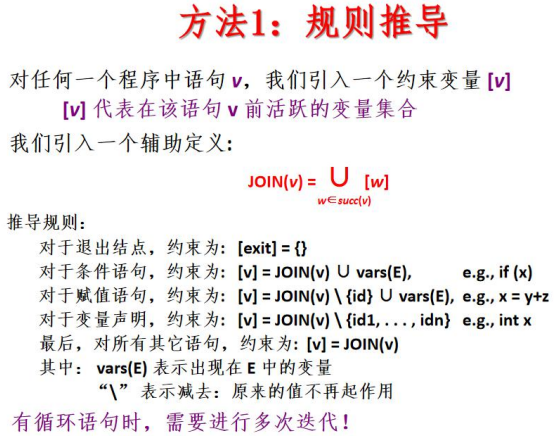
基本方法是不动点算法，即一直进行活性逆向传递，直到所有语句的活性都不再改变。对应的代码如下：

```
while(changed) {
```

```
                changed = false;
                for(int i=proc.stmtlen-1;i>=0;--i) {
                    Node stmt = n.f0.nodes.elementAt(i);
                    MyStmt mystmt = proc.stmtlist.elementAt(i);
                    oldlives.clear();
                    oldlives.putAll(mystmt.livevars);
                    stmt.accept(this, mystmt);
                    changed                                    =
(mystmt.liveschanged(oldlives))?true:changed;
                }
            }
```

其中 oldlives 的作用只是用于和传递之后的活跃变量列表进行比对，以判断是否到达了不动点。

对于具体的语句，我使用了规则推导的方法来计算活性。每一种的计算规则已经在课件中讲的很清楚了（如下图），因此不再赘述。



4.3 寄存器分配

关于寄存器分配，老师介绍了两种最常见的方法：基于图着色的方法和基于线性扫描的方法。

我提交的第一个版本使用了线性扫描的方法，但是由于我没有充分理解该算法的含义，造成了冗余。我是在每一个基本块内进行寄存器分配，每次进入和离开块时需要将所有的活跃变量从栈中读入或者写出，所以产生了大量的冗余指令。因此，经过查阅文献，我发现基础的线性扫描算法并不需要考虑活跃变量和基本块，只需要将变量出现的最远区间作为活性区间，然后通过线性扫描的方法进行寄存器分配。但是这样的简单线性扫描 live interval 方法自然不如基于每个语句的活性分析结果进行寄存器分配算法的效率高，但是更高级的扫描 live range 的方法我还没有读懂，因此我选择了使用更经典、寄存器利用率更高的图着色算法作为优化。

图着色算法需要构造一个干扰图——以 temp 变量为结点，以在一个语句同时活跃作为关系——，对于干扰图进行着色。

着色的颜色数量设置为 K=18，即包括了 s 寄存器和 t 寄存器，而剩下的 a 寄存器专门用作参数传递，v0 寄存器用于返回值传递，v0 和 v1 寄存器用于不活跃的 temp 变量的使用。

这一部分是在 visitor.InterferenceGraph.java 中实现，特别是其中的 _initproc 方法涵盖了各个子步骤的调用。

(1) 首先初始化过程的 temps 列表。遍历所有的 Temp 结点，将其和一个 MyTemp 对应，放入 temps 索引表（若之前出现过，则跳过）。

(2) 构造干扰图。干扰图的数据结构，是 temps 列表中的每一个

MyTemp 变量的 interference 列表，记录了与该 temp 变量互相干扰的变量。（类似于图的邻接表表示方法）

(3) 着色和寄存器分配。使用简单的启发式着色方法：每次删去度数小于 K 的结点，直到结点数为 0 或者无法删除（则溢出然后继续）。而实现的数据结构，包括了优先队列（结点关于度数的最小堆，每次删去堆头）、栈（删去的结点放入到栈中，之后着色是从栈顶开始）、列表（记录溢出变量，并分配溢出单元）。

(4) 溢出。如上，每次没有度数小于 K 的结点时，溢出一个结点，然后再次尝试删去，重复操作直到全部都删去或者溢出。对于溢出单元的编号，放在栈中的过程实参（Temp4, Temp5, ...）是编号最小的溢出单元，必须按照参数的顺序来访问；其他的溢出单元就只要有编号就可以。

4. 4 翻译生成

完成了上述三个部分之后，翻译生成的工作就变得很简单了。

Spiglet 和 kanga 的语法结构有一定相似处，大多数 spiglet 语句都可以找到对应的一个语句，因而只需要将出现的 temp 变量替换为 mytemp 对象记录的寄存器即可。有以下四点需要注意：

(1) 溢出的 temp 变量，需要先增加一个”ALOAD v0/v1 SPILLEDARG I”语句，然后再将 temp 变量替换为 v0/v1.

(2) 过程翻译开始前，先将实参加载。如果是前 4 个实参，则需要从 a 寄存器”MOVE”” 到对应的寄存器或者”ASTORE”到对应的溢出单

元；如果是之后的实参，本身就放在了溢出单元，只有当有分配的寄存器时，才需要”ALOAD”到对应的寄存器。

(3) Kanga 的 Call 单独成一条指令。因此需要先将参数传递到 a 寄存器或者溢出单元中，然后再生成 call 指令，之后将原来的”MOVE temp Call ...”翻译成”MOVE reg v0”。

(4) kanga 没有 RETURN 语句，因此需要在过程最后添加一个”MOVE v0 ...”或者”ALOAD v0 ...”之类的语句，将返回值专门拷贝到 v0 寄存器中。