编译实习期末报告

1000010284 管毓清

2012 年 12 月 1 日

目录

一、概述

1. 编译器基础知识

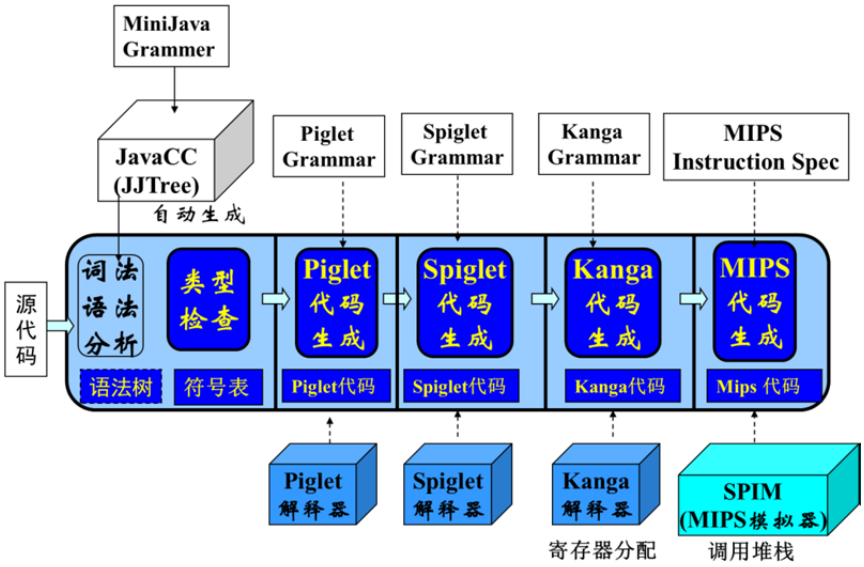
编译器(Compiler)，是一种电脑程序，它会将用某种编程语言写成的源代码（原始语言），转换成另一种编程语言（目标语言）。它主要的目的是将便于人编写，阅读，维护的高级计算机语言所写作的源代码程序，翻译为计算机能解读、运行的低阶机器语言的程序，也就是可执行文件。一个现代编译器的主要工作流程如下： 源代码 (source code) → 预处理器 (preprocessor) → 编译器 (compiler) → 汇编程序 (assembler) → 目标代码 (object code) → 链接器 (Linker) → 可执行文件 (executables)[1]

2. 编译器基本功能

编译实习作业一共分为五步：

1) 类型检查: 主要要做的任务是建立符号表，检查类型错误（未定义、重复定义的类、变量、方法，参数个数和类型是否匹配，判断表达式类型以及操作数相关类型的检查）；
2) MiniJava→Piglet:修改符号表，添加各类信息，翻译成 Piglet 中间代码，去除类，使用临时单元，处理继承关系；
3) Piglet→Spiglet:消除嵌套表达式，转换为接近三地址代码的 SPiglet 语言；
4) Piglet→Kanga:临时单元活性分析，寄存器分配，代码优化；
5) Kanga→MIPS Assembly:映射为机器指令，指令选择，维护堆栈，生成汇编代码。

整个过程如下：



要求在 Java 平台上实现这五个步骤，将给定的 MiniJava 代码编译成 MIPS 汇编代码，能在 SPIM 模拟器上运行并输出正确结果。

3. 编译器特点

本作业较好的完成要求的五个步骤（在五次测试中均在最高的分数区间(14~15)内），程序运行时对对象、数组的访问进行了空指针、越界等安全性检查，可靠性强；使用了流图分析和静态单赋值形式，消除了死代码和不可能执行到的代码，并优化了线性扫描算法，减少了寄存器的使用和代码长度、运行时间。

注：由于本报告写于王老师发布实习报告参考框架前，一开始写的报告与参考框架差异过大，之后针对参考的框架进行了一些修改，但是仍然存在一定的差异，如编译器设计与实现写在一起。对工具软件的介绍移至报告的第三部分中，请见谅。

[1]维基百科-编译器 http://zh.wikipedia.org/wiki/编译器

二、编译器设计与实现

1. 类型检查

1.1. 任务要求

在本步骤之前 ,输入的 MiniJava(Java 的一个子集 ,将会在 2.2 中介绍)已经过 MiniJavaParser 语法分析，确定其没有语法错误。但是仍然可能有语义错误，这些错误的存在会影响到中间代码翻译。而类型错误即是语义错误中的一种。本步骤要求对如下错误进行检查：

1) 使用未定义的类、变量、方法
2) 重复定义了类、变量、方法
3) 用于判断的表达式必须是 boolean 型
4) 操作数相关：+、*、< 等的操作数必须为整数
5) 数组下标不是整数，创建数组时数组长度不是整数
6) 方法参数类型不匹配，参数个数不匹配，return 语句返回值类型与方法声明的返回值类型不匹配
7) 类的循环继承
8) 不允许多继承（类 C 直接继承 B 和 A）（不用检查，MiniJava 的 BNF 范式已经保证了这一点）
9) 不允许方法重载（即一个类中定义若干同名的方法）（等同于检查重定义错误）

难度系数：4

1.2. MiniJava 简介

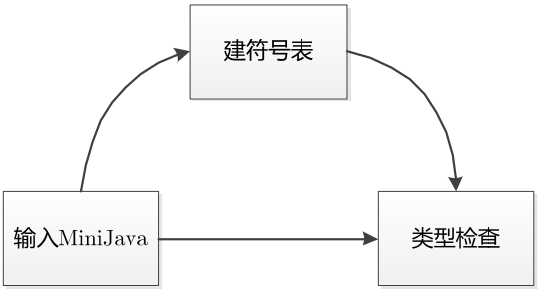
MiniJava 是 Java 的子集，缺省约定遵从 Java，但做了一些额外的精简约定：

1) 不允许方法重载(Overloading)
2) 类中只能申明变量和方法（不能嵌套类）
3) 只有类，没有接口，有继承关系（单继承）
4) 只有四种类型的变量：整型(int)，布尔型(boolean)，整型数组(int[])，对象(程序中定义的类)
5) 二元操作符只有:&&,+,<,*，&&左右必须为布尔型变量，其他操作符左右必须为整型变量
6) 主函数有且只有一条语句，内容为 System.out.println(Exp); Exp 为表达式
7) 所有方法必须为 public 方法，必须有返回值
8) 创建对象时不能使用参数，也不能有构造函数
9) 类中成员变量声明必须写在方法声明前，方法中局部变量声明必须写在语句(Stmt)前
10) 类中成员变量/方法中局部变量声明时均不可以初始化
11) 语句(Stmt)共有六种，分别是代码块(Block)，赋值语句(AssignmentStatement)，数组元素赋值语句(ArrayAssignmentStatement)，条件语句(IfStatement)，循环语句(WhileStatement)，打印语句(PrintStatement)
12) 打印语句只能打印整型变量的表达式
13) 表达式(Exp)共九种：短路与表达式(AndExpression)，小于比较表达式(CompareExpression)，加法表达式(PlusExpression)，减法表达式(MinusExpression)，乘法表达式(TimesExpression)，数组元素表达式(ArrayLookup)，数组长度表达式(ArrayLength)，方法调用表达式(MessageSend)，基本表达式(PrimaryExpression)；其中基本表达式也有九种：数字(IntegerLiteral)，逻辑真(TrueLiteral)，逻辑假(FalseLiteral)，标识符(Identifier)，this 指针(ThisExpression)，创建数组(ArrayAllocationExpression)，创建对象(AllocationExpression)，逻辑非表达式(NotExpression)，括号表达式(BracketExpression)。

1.3. 任务分析

类型检查的核心任务是建立符号表，通过符号表检测类型错误是否存在，而且此符号表也能为以后中间代码翻译使用。对于建立符号表/类型检查，可以采取两种不同的方法：通过一个 visitor 一次扫描整个语法树，一边建立符号表一边检查类型错误。这种方法的优点是速度较快，但是不利于编写和维护。特别是对于如变量声明在前，而其类型声明在后这类情况，处理较为复杂。另一种方法是通过两个 visitor，第一次扫描建立符号表并进行一部分已经可以完成类型检查，第二次扫描完成剩余的类型检查工作。这种方法运行起来不如前一种块，但是代码更加直观，更利于编写、调试和维护。

我的作业中采用的就是第二种方法，第一次扫描代码，使用了 BuildSymbolTableVisitor 建立符号表并进行标识符（类名、方法名、变量名）的判重工作。第二次扫描，使用 TypeCheckVisitor 检查其他的类型错误（以及在第一次扫描时不能完成的判重工作，主要是子类方法对父类方法出现重载）。



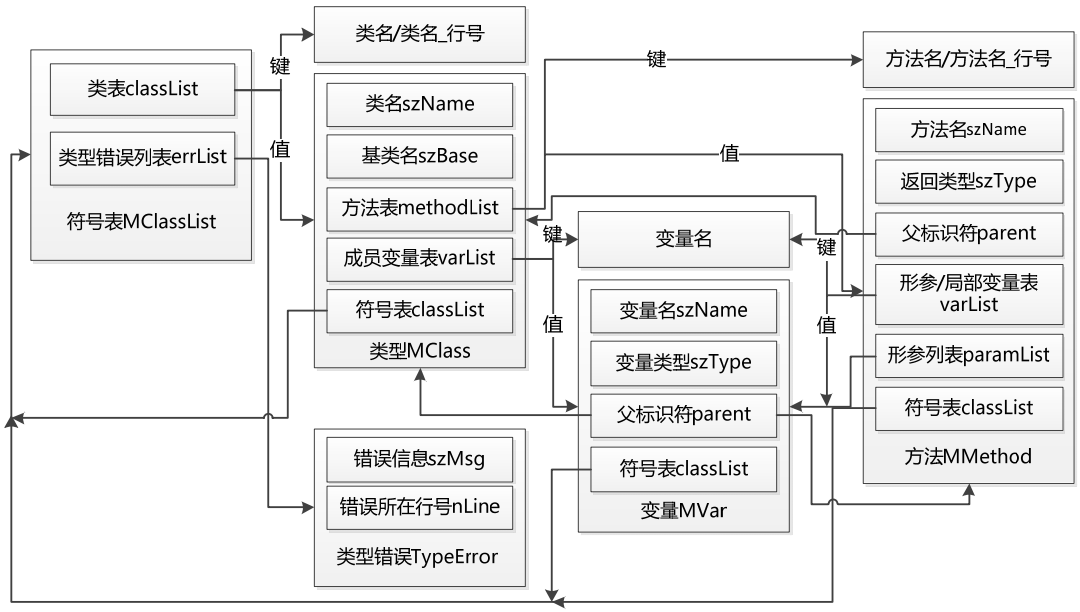
1.4. 符号表结构

对 MiniJava 建立的符号表，即将标识符与对应的类型建立起联系。所有的标识符分为三种：

1) 类
2) 方法
3) 变量

相对应的，建立起三个不同的类，MClass，MMethod，MVar，代表每一个声明的类、方法和变量均派生自 MIdentifier 抽象基类，同时建立起总符号表 MClassList 保存所有类，针对 int, boolean, int[]三种类型，还从 MClass 派生出了 MbasicType 类。为了便于检测实参列表，还建立了 MActualParamList 类。

符号表主要的类关系如下：



详细解释：

MClassList 类：总符号表，保存所有的类

MClassList 类成员变量（与类型检查有关的）：

类型	变量名	功能
Hashtable<String, MClass>	classList	类名到类的映射，类列表
Vector<TypeError>	errList	整个符号表中将产生的类型错误列表

MClassList 类成员方法（与类型检查有关的）：

类型	方法名	参数	功能
N/A	MClassList	N/A	构造函数，预先插入整型、布尔、数组三种基本类型
boolean	insert	newClass：要插入的类 nLine：行号	插入新类，返回是否重定义 其中，如果发生重定义，新类在 classList 中的键值将会更改为：原类名_行号
MClass	get	szName：类名	不通过行号直接获取类
MClass	get	szName：类名 nLine：行号	通过行号和类名获取类，通过这种方法，可以进入重定义声明的类，检查其内部是否有类型错误
boolean	contains	szName：类名	是否存在此类
boolean	classEqualsOrDerives	szBase：A 类名 szExt：B 类名	检查 A 类是否为 B 类父类或者就是 B 类
void	addError	e：类型错误	加入一个通过符号表检查到的类型错误
void	printErrors	N/A	打印所有错误
boolean	hasTypeErrors	N/A	检查后是否有类型错误
String	toString	N/A	将类、方法、变量的层次结构输出

MClass 类：类型

MClass 类成员变量（与类型检查有关的）：

类型	变量名	功能
String	szName	类名
MClassList	classList	符号表
String	szBase	基类名
Hashtable<String, MMethod>	methodList	方法名到方法的映射
Hashtable<String, MVar>	varList	变量名到成员变量的映射

MClass 类成员方法（与类型检查有关的）：

类型	方法名	参数	功能
void	setBase	szBase：基类名	设置基类名
String	getBase	N/A	获取基类名

MClass 类成员方法（与类型检查有关的，续表）：

类型	方法名	参数	功能
boolean	insertMethod	newMethod：方法 nLine：行号	插入新方法，返回是否重定义 若发生重定义，新方法在 methodList 中的键值将更改为：原方法名_行号
MMethod	getMethod	szMethod：方法名	不通过行号直接获取方法
MMethod	getMethod	szMethod：方法名 nLine：行号	通过行号和方法名获取方法，通过这种方法，可以进入重定义声明的方法，检查其内部是否有类型错误
MMethod	getMethodInBase	szMethod：方法名	在基类中查找同名方法
boolean	insertVar	newVar：新变量	检测变量是否重复定义，否则插入
MVar	getVar	szName：变量名	获取成员变量
MVar	getVarInBase	szName：变量名	在基类中查找变量
String	toString	N/A	将方法、变量的层次结构输出
boolean	isBasicType	N/A	是否是 int, boolean, int[]三种基本类型

MMethod 类：方法

MMethod 类成员变量（与类型检查有关的）：

类型	变量名	功能
String	szName	类名
String	szType	返回类型
MIdentifier	parent	父标识符（所在的类）
MClassList	classList	符号表
Hashtable<String, MVar>	varList	变量名到形参/局部变量的映射
Vector<MVar>	paramList	形参列表

MMethod 类成员方法（与类型检查有关的）：

类型	方法名	参数	功能
boolean	insertParam	newParam：新参数	插入形参，再调用 insertVar 尝试插入变量
boolean	insertVar	newVar：新变量	检测变量是否重复定义，否则插入
MVar	getVar	szName：变量名	获取局部变量，若找不到，则搜寻父标识符
String	toString	N/A	将类、方法、变量的层次结构输出
boolean	equals	o：另一个方法	判断 o 方法的返回类型是否等同于自己的返回类型或者是其子类 判断 o 方法的形参个数和类型是否与自己的形参个数和类型严格相等
int	checkParam	p：实参列表	检查实参列表 返回 0：实参和形参匹配 返回 1：参数个数不匹配 返回 2：参数类型不匹配

MVar：变量

MVar 类成员变量（与类型检查有关的）：

类型	变量名	功能
String	szName	类名
String	szType	返回类型
MIdentifier	parent	父标识符（所在的类）
MClassList	classList	符号表

MVar 类成员方法（与类型检查有关的）：

类型	方法名	参数	功能
String	toString	N/A	输出变量定义

1.5. 建立符号表

使用了 BuildSymbolTableVisitor 类建立符号表，建立符号表的时候主要要考虑标识符的声明：

1) 类声明

在类声明中获取类名，如果有基类的话，设置基类名。判断在符号表中是否重定义，然后插入符号表。

a) 主类声明

```
public String visit(MainClass n, MType argu)//主类
{
     n.f0.accept(this, classList);

     String szClass = n.f1.accept(this, classList);//主类名
     MClass newClass = new MClass(szClass);//创建主类

     classList.insert(newClass, n.f1.f0.beginLine);//直接插入主类

     n.f2.accept(this, newClass);//主类内部
     n.f3.accept(this, newClass);
     n.f4.accept(this, newClass);
     n.f5.accept(this, newClass);
     n.f6.accept(this, newClass);

     MMethod newMethod = new MMethod("main", "void", newClass);//创建main方法
     newClass.insertMethod(newMethod, n.f1.f0.beginLine);//插入main方法

     n.f7.accept(this, newMethod);//main方法内部
     n.f8.accept(this, newMethod);
     n.f9.accept(this, newMethod);
     n.f10.accept(this, newMethod);

     String szVar = n.f11.accept(this, newMethod);//方法参数
     MVar newVar = new MVar(szVar, "String []", newMethod);//参数变量
     newMethod.insertParam(newVar);//插入参数

     n.f12.accept(this, newMethod);
     n.f13.accept(this, newMethod);
     n.f14.accept(this, newMethod);
     n.f15.accept(this, newMethod);
     n.f16.accept(this, newClass);

     return null;
}
```

由于主类的声明中存在一个 main 方法的声明，这个方法也同时插入主类。main 方法的声明中也存在一个形参的声明，这个形参将插入到主类的 main 方法中。

由于 BNF 范式中已经规定了只有一个主类，主类中只有一个 main 方法，main 方法只有一个形参，无成员变量。所以在此处声明不考虑任何重复定义的检查。

b) 不含基类的类声明

```
    public String visit(ClassDeclaration n, MType argu)//类声明(不含基类)
    {
        n.f0.accept(this, classList);

        String szClass = n.f1.accept(this, classList);//类名
        MClass newClass = new MClass(szClass);//创建类

        if (!(classList).insert(newClass, n.f1.f0.beginLine))//插入符号表，判重
        {
            classList.addError(new TypeError(n.f1.f0.beginLine, "Multiple class declarations:
" + "\"" + szClass + "\""));
        }

        n.f2.accept(this, newClass);//类内部
        n.f3.accept(this, newClass);
        n.f4.accept(this, newClass);
        n.f5.accept(this, newClass);

        return null;
    }
```

如果已经有同类名的类，则产生一个重复定义错误，但是仍然插入（键值改为：类名_行号）。

c) 含基类的类声明

```
    public String visit(ClassExtendsDeclaration n, MType argu)//类声明（含基类）
    {
        n.f0.accept(this, classList);

        String szClass = n.f1.accept(this, classList);//类名
        MClass newClass = new MClass(szClass);//创建类

        if (!(classList).insert(newClass, n.f1.f0.beginLine))//插入符号表，判重
        {
            classList.addError(new TypeError(n.f1.f0.beginLine, "Multiple class declarations:
" + "\"" + szClass + "\""));
        }

        n.f2.accept(this, newClass);//类内部

        String szBase = n.f3.accept(this, newClass);//基类名
        newClass.setBase(szBase);//设置基类

        n.f4.accept(this, newClass);
        n.f5.accept(this, newClass);
        n.f6.accept(this, newClass);
        n.f7.accept(this, newClass);

        return null;
    }
```

如果已经有同名的类，则产生一个重复定义错误，但是仍然插入（键值改为：类名_行号）。设置基类。

2) 方法声明

在方法声明中获取方法名，并且判断在符号表中是否重定义，然后插入符号表。

```
    public String visit(MethodDeclaration n, MType argu)//方法声明
    {
        n.f0.accept(this, argu);

        String szType = n.f1.accept(this, argu);//方法类型
        String szMethod = n.f2.accept(this, argu);//方法名

        MMethod newMethod = new MMethod(szMethod, szType, (MClass) argu);//新方法
        if (!((MClass) argu).insertMethod(newMethod, n.f2.f0.beginLine))//插入父类,判重
        {
            classList.addError(new TypeError(n.f2.f0.beginLine, "Multiple methods
declarations: " + "\"" + ((MIdentifier) argu).getName() + "." + szMethod + "\""));
        }
```

```
        n.f3.accept(this, newMethod);//方法内
        n.f4.accept(this, newMethod);
        n.f5.accept(this, newMethod);
        n.f6.accept(this, newMethod);
        n.f7.accept(this, newMethod);
        n.f8.accept(this, newMethod);
        n.f9.accept(this, newMethod);
        n.f10.accept(this, newMethod);
        n.f11.accept(this, newMethod);
        n.f12.accept(this, newMethod);

        return null;
    }
```

如果已经有同名的方法，则产生一个重复定义错误，但是仍然插入（键值改为：方法名_行号）。

3) 变量声明

a) 成员变量/局部变量声明

```
    public String visit(VarDeclaration n, MType argu)//变量声明
    {
        String szType = n.f0.accept(this, argu);//类型
        String szVar = n.f1.accept(this, argu);//变量名

        MVar newVar = new MVar(szVar, szType, (MIdentifier) argu);//创建变量
        if (!((MIdentifier) argu).insertVar(newVar))//插入父标识符,判重
        {
            classList.addError(new TypeError(n.f1.f0.beginLine, "Multiple variables
declarations: " + "\"" + szVar + "\""));
        }

        n.f2.accept(this, argu);

        return null;
    }
```

如果已经有同名的变量，则产生一个重复定义错误，不再插入。

b) 形参声明

```
    public String visit(FormalParameter n, MType argu)//参数声明
    {
        String szType = n.f0.accept(this, argu);//参数类型
        String szVar = n.f1.accept(this, argu);//参数名

        MVar newVar = new MVar(szVar, szType, (MIdentifier) argu);//创建变量
        if (!((MMethod) argu).insertParam(newVar))//参数判重
        {
            classList.addError(new TypeError(n.f1.f0.beginLine, "Multiple parameters
declarations: " + "\"" + szVar + "\""));
        }

        return null;
    }
```

如果已经有同名的参数，则产生一个重复定义错误，只插入参数列表不插入变量列表。

注意：

建立符号表时可以进行重复定义检查，但是有一种重复定义还不能检查：子类方法对父类方法的重复定义，即子类方法和父类方法重名，且它们之间的关系不是覆盖而是重载。这必须得等到所有父类方法都添加到符号表后才能检查。

对于重复定义的类，方法，本作业的原则是，仍然要对其内部进行类型检查。所以也要把它们添加到符号表中，并且向其内部添加方法与变量。对于重复定义了的类和方法，在类名/方法名到类/方法的映射的键值为名称_行号。由于 MiniJava 不可能在同一行内定义两个类或方法，这样就保证了类，方法的唯一性。

1.6. 检查类型错误

1) 使用未定义的类、变量、方法

出现未定义标识符的时机有：

语法树节点	错误类型
ClassExtendsDeclaration	未定义的基类
VarDeclaration	未定义的变量类型
MethodDeclaration	未定义的返回类型
FormalParameter	未定义的参数类型
AssignmentStatement	未定义的被赋值变量
ArrayAssignmentStatement	未定义的数组
MessageSend	未定义的方法
AllocationExpression	未定义的类型
PrimaryExpression[1]	未定义的变量

对于未定义类的检查，是在总的符号表中检查是否有同名的类，没有则产生一个未定义类错误。

对于未定义方法的检查，是检测方法是否在给定的类 A 中，如果在类 A 不存在，则检查 A 的基类，重复此过程，直到无基类/基类不存在（未定义类错误）/基类在之前已经检查过(循环继承错误)。

对于未定义变量的检查 ,先检查变量是否是当前方法的参数/局部变量 ,然后再查找当前方法所在的类，在此类和其基类中查找，与检查未定义方法类似。

2) 类的循环继承

```
    String szBases = newClass.getName();
    HashSet<String> lstBase = new HashSet<String>();
    lstBase.add(newClass.getName());

    while (szBase != null)
    {
        szBases += "->" + szBase;
        if (szBase.equals(newClass.getName()))//出现循环继承
        {
            szBases = szBases.substring(0, szBases.length());
            classList.addError(new TypeError(n.f3.f0.beginLine, "Circular extends: " + "\""
+ szBases  + "\""));
            break;
        }
        else if (lstBase.contains(szBase)) break;

        lstBase.add(szBase);

        MClass baseClass = classList.get(szBase);
        if (baseClass != null) szBase = baseClass.getBase();
        else break;
    }
```

以访问含基类的类 A 定义为例：先将 A 类本身加入集合 lstBase，然后循环加入 A 类的基类/A 类的基类的基类……一直到找到的 B 类无基类、基类未定义，或者基类就是 A 类，或者基类不是 A 类但是已经在 lstBase 中。第一种情况是正常情况，第二种情况是未定义类错误，但是已在 B 类的声明中检查并输出了错误。第三种情况就是 A 类出现在了循环继承中，需要输出循环继承错误；第四种情况是 A 类未出现在循环继承中，但是其基类中出现了循环继承，这个时候会由 A 类的处在循环内的基类定义处输出错误，所以在这里也不输出错误。

[1] 由于 PrimaryExpression 本身没有提供自己所在行号的属性，为保证所有类型错误皆有其对应的行号，本作业中，对于 PrimaryExpression 的未定义变量错误实际上是在使用了它或者使用 Expression 非终结符号的产生式的 visit 函数中检查的。

3) 重复定义了类、变量、方法

上面已经说了，在建立符号表时可以对绝大多数重复定义错误进行检查，但是不可以检查子类方法对父类方法的重载。

在每一个方法定义处，调用 MClass 的 getMethodInBase 查找基类中是否有与当前方法同名的方法，如果没有，则不可能出现重复定义。

如果有，比较两个方法：假设基类为 A，子类为 B，方法名均为 c，A 类和 B 类的 c 方法（A.c, B.c）返回类型分别为 D,E，则 B.c 是 A.c 的覆盖而非重载当且仅当：A.c 和 B.c 的参数个数相等，所有参数类型严格一致。返回类型，E 要么就是 D，要么是 D 的子类。（第一次面测时，我和助教老师通过 Eclipse 的错误提示功能，已经证实了这一点）

```
public boolean equals(Object o)//MMethod类方法，判断是继承还是重载
{
    MMethod other = (MMethod) o;
    if (!classList.classEqualsOrDerives(other.szType, szType)) return false;
    //返回类型必须兼容
    int nParam = paramList.size();
    if (nParam != other.paramList.size()) return false;//参数个数一致
    for (int i = 0; i < nParam; ++i)//参数类型一致
        if(!paramList.get(i).getType().equals(other.paramList.get(i).getType()))
            return false;
    return true;
}
```

4) 用于判断的表达式必须是 boolean 型，操作数相关：+、*、< 等的操作数必须为整数，数组下标不是整数，创建数组时数组长度不是整数，return 语句返回值类型与方法声明的返回值类型不匹配

上述了很多种类型错误，本质上都是类型不匹配错误，在类型检查的 Visitor 中，每一个 visit 函数返回值都是一个 MIdentifier，包含了类型信息，对于所有的 Expression 或者 PrimaryExpression，visit 函数将返回其对应的类型。这样在需要用到类型匹配的时候，直接检测 visit 函数的返回的类型是否正确即可。

与此同时，为了保证一个错误只产生一条提示（如 int a; a = b;//b 未定义，只产生未定义错误，不产生额外的类型匹配错误），已经包含了此类错误的表达式的类型将置为 NULL，以便不再做额外的类型匹配检查。（除非是参数匹配，或者类型非常明显的 int a; a = !b;//虽然 b 未定义，但是显然!b 是一个 boolean，所以仍然要报匹配错误）

例子：&&运算的类型，检查左操作数是否为 boolean

```
MIdentifier exp = n.f0.accept(this, argu);//获取表达式
if (exp != null)//表达式不含(但可能是)未定义变量
{
    String szExpType = exp.getType();//表达式类型

    if (szExpType == null)//表达式是未定义变量
    {
        classList.addError(new TypeError(n.f1.beginLine, "Undefined variable: " + "\"" +
exp.getName() + "\""));
    }
    else
    {
        if (!exp.getType().equals("boolean"))//类型不匹配
        {
            classList.addError(new TypeError(n.f1.beginLine, "Left expression of '&&' is
not a boolean"));
        }
    }
}
```

5) 方法参数类型不匹配，参数个数不匹配

建立一个实参列表类 MActualParamList，每次在访问 MessageSend 时记录所有的实参，然后先检测实参是否有未定义错误，然后调用 MMethod 的 checkParam 函数，先检测参数个数是否匹配，然后检查是否有实参为未定义标识符、或者实参类型既不等于形参类型，也不是形参类型的子类。

2. MiniJava → Piglet

2.1. 任务要求

在类型检查中已经建立起了 MiniJava 代码的符号表，在这一步中，需要根据符号表，将 MiniJava 代码转换为 Piglet 代码，去除所有的类，所有的成员方法都变为全局过程，所有的变量都变为临时单元，或者变为临时单元指向的一个内存区域。

重点：分配临时单元给成员、局部变量和参数；处理类继承关系与方法覆盖；创建对象、创建数组、访问数组、对象内容、调用对象方法的实现。

难度系数：3

2.2. Piglet 语言介绍

Piglet 是一种接近中间代码的语言，采用前缀表达式，操作符在最前面，比一般的中间代码抽象层次高，表达上接近源语言，语句中允许包含复杂的表达式，不是严格的三地址码。它的 BNF 范式为：

Goal ::= **"MAIN" StmtList "END" (Procedure)* <EOF>**
StmtList ::= **((Label)? Stmt)***
Procedure ::= **Label "[" IntegerLiteral "]" StmtExp**
Stmt ::= **NoOpStmt**
|**ErrorStmt**
|**CJumpStmt**
|**JumpStmt**
|**HStoreStmt**
|**HLoadStmt**
|**MoveStmt**
|**PrintStmt**
NoOpStmt ::= **"NOOP"**
ErrorStmt ::= **"ERROR"**
CJumpStmt ::= **"CJUMP" Exp Label**
JumpStmt ::= **"JUMP" Label**
HStoreStmt ::= **"HSTORE" Exp IntegerLiteral Exp**
HLoadStmt ::= **"HLOAD" Temp Exp IntegerLiteral**
MoveStmt ::= **"MOVE" Temp Exp**
PrintStmt ::= **"PRINT" Exp**
Exp ::= **StmtExp | Call | HAllocate | BinOp | Temp | IntegerLiteral | Label**
StmtExp ::= **"BEGIN" StmtList "RETURN" Exp "END"**
Call ::= **"CALL" Exp "(" (Exp)* ")"**
Hallocate ::= **"HALLOCATE" Exp**
BinOp ::= **Operator Exp Exp**
Operator ::= **"LT" | "PLUS" | "MINUS" | "TIMES"**
Temp ::= **"TEMP" IntegerLiteral**
IntegerLiteral ::= **<INTEGER_LITERAL>**
Label ::= **<IDENTIFIER>**

可以看出，在 Piglet 语言中，已经不再存在类、类型等概念。所有的变量被 TEMP 所取代，其中 TEMP 0~TEMP 19 为各过程参数。其它的临时单元可以随意分配给各局部变量，以及表达式计算的临时结果。

下面着重介绍一下 Piglet 语言中数组和对象的存储方式。

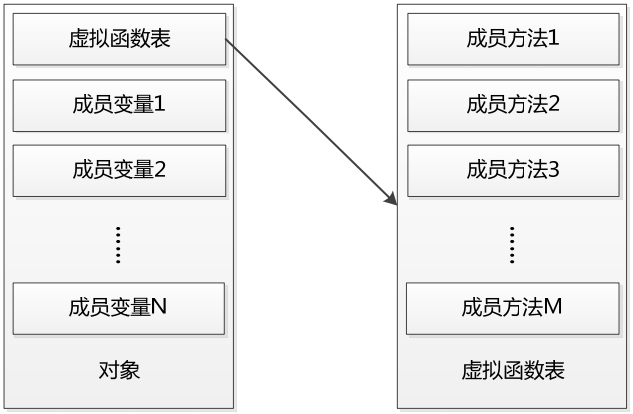
总所周知，Java 的数据类型分为基本类型和引用类型。前者有 int, boolean, float, char 等，直接保存在栈中。后者在堆中分配一块内存空间，在栈中保留一个指针。体现在 MiniJava 至 Piglet 的转换中，即 int, boolean 两种基本类型均直接保存在临时单元中，而 int[]或者对象类型都是在堆中分配空间，然后把其地址保存在临时单元中。

对于一个数组 A，结构如下：

0	**1**	**2**	**……**	**Length - 1**	**Length**
Length	A[0]	A[1]	……	A[Length - 2]	A[Length - 1]

假设数组 A 长度为 L，则必须开辟 4*L+4 的空间，前四个字节保存数组长度，然后每四个字节保存一个数组元素。且在创建数组时，应该将数组每一个元素均赋初值为 0.

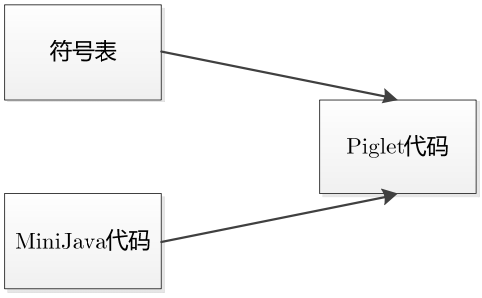
对于一个对象来说，也可以将其看成一个数组。从第二个元素开始到结束每四个字节保存一个成员变量。而前四个字节又指向另外一个数组，保存这个对象所有的方法。这么做是为了实现子类函数对基类函数的覆盖，同样是基类引用，调用同样名称的方法。如果引用指向基类对象，则调用基类方法，如果引用指向某一种子类对象，则调用这种子类的同名方法。每一个对象绑定一个虚拟函数表，以实现多态。



2.3. 任务分析

在建立好符号表之后，我们已经可以对每一个类的成员变量和方法，分配其在这个类的对象的内存空间中的位置。与此同时，我们也可以提前对每一个方法的参数和局部变量分配好所用的临时单元，减少下一次扫描代码时的工作量。在这一步，我们着重要考虑类的继承关系，如在子类中怎么保证覆盖掉父类的方法，子类对基类的成员变量、成员方法的访问问题。

做完这些准备工作后，我们就可以开始扫描代码，在语法制导翻译方案中，将 MiniJava 代码转成 Piglet 代码。在 MiniJava 中，代码是以类为核心来组织的，所有的语句都是在类之中。而 Piglet 和随后的几种中间代码，都是以过程为核心而组织的，所有语句依托于过程的存在而存在。面向对象的成员方法必须改写为面向过程的全局过程。在转换的同时，为了数据安全，本作业添加了对数组、对象访问的空指针引用检查，以及对数组定义时长度有效性，访问数组时下标越界等都做了检查。



2.4. 存储空间分配

由于 Piglet 语言中没有了类型的概念，所有的临时单元（整型，布尔型，指针）都占据 4 个字节大小。假如不考虑类继承，对于每一个类 MClass，就是从 varList 中取出各个变量，分别设置位置为 4，8，12…。然后从 methodList 中取出各个方法，设置其在虚拟函数表中的位置分别为 0，4，8…

对于每一方法，假如不考虑其参数超过 19 个的情况，从第一个参数开始分配 TEMP 1，2，3…TEMP 0 分配给这个方法所在的对象。从 TEMP 20 开始依次给所有局部变量分配一个临时单元即可。

2.5. 类的继承和方法覆盖

在 3.4 节中我们假设不考虑类继承，如果引入了类的继承。对于子类和父类，存在方法的覆盖现象，而子类方法也可以调用父类的方法和成员变量。甚至子类和父类可以重复定义相同名称的变量互不影响。（这不是类型错误）。

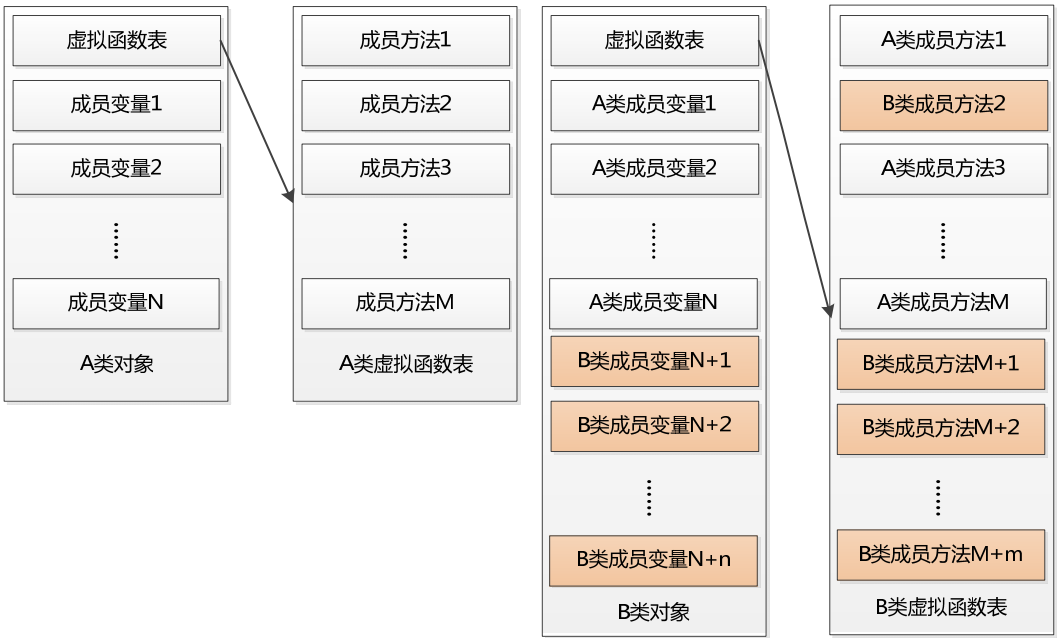
但是我们知道，一个子类的对象可以赋值给一个父类的引用，例如，假设 A 是 B 的基类，我们可以写出这样的语句：

```
A b = new B(); b.fun();
```

假设 A 类和 B 类都定义了 fun 方法，B.fun 会覆盖掉 A.fun 方法，但是 b 是一个 A 类的引用，按照默认情况，解释器会去查找 A 类对象的虚拟函数表，找到 fun 函数在其内存空间中存放的位置。为了保证 B 类方法能被正确调用，唯一的办法是将 B.fun 和 A.fun 保存到同一个位置。

同样，假设 A 类中有一个成员变量 c，B.fun 方法访问了这个变量，A.fun1 方法（B 类中无 fun1）也访问了这个变量。那么当执行 b.fun()和 b.fun1()时，显然 B 类中的 c 位置是固定的，那么 B 类方法 fun 和 A 类方法 fun1 都必须在同一个位置来读取/写入变量 c，也就是说基类 A 对象的成员变量（和方法）和子类 B 对象中的 A 类的成员变量（和方法）必须处于同一个位置。

所以，在为成员函数和变量分配存储位置时，应该遵循先基类/后子类的原则，从最上层无基类的顶层基类开始，依次分配虚拟函数表和成员变量的位置，且子类方法要替换掉父类的同名方法。（无重载）



上图就是假设 A 类是 B 类基类的情况，其中 B 类的成员方法 2 覆盖掉了 A 类的成员方法 2。

下面是设置成员变量/方法偏移位置的函数代码：

```
    private void setOffsets(Hashtable<Integer, MMethod> methodOffsets, Hashtable<Integer, MVar>
varOffsets)//设置偏移
    {
        if (szBase != null)
        {
            MClass baseClass = classList.get(szBase);
            if (baseClass != null) baseClass.setOffsets(methodOffsets, varOffsets);
            //先在基类找
        }

        int nSize, nMethodSize, nBaseMethodSize;

        nSize = varOffsets.size() * 4 + 4;
        nMethodSize = nBaseMethodSize = methodOffsets.size() * 4;

        for (MVar _var : varList.values())//加入各变量
        {
            _var.setOffset(nSize);
            varOffsets.put(nSize, _var);
            nSize += 4;
        }

        for (MMethod _method : methodList.values())//加入各方法
        {
            boolean bFlag = true;
            for (int i = 0; i < nBaseMethodSize; i += 4)
            {
                if (methodOffsets.get(i).getName().equals(_method.getName()))
                {
                    _method.setOffset(i);
                    methodOffsets.put(i, _method);//覆盖掉基类方法

                    bFlag = false;
                    break;
                }
            }

            if (bFlag)//没有覆盖，加入新方法
            {
                _method.setOffset(nMethodSize);
                methodOffsets.put(nMethodSize, _method);
                nMethodSize += 4;
            }
        }
    }
```

需要注意的是，与 Java 实际执行的情况一样，只有当创建一个新对象时，才会按照上述函数设置好的偏移位置，依次向对象开辟的内存空间中存入各方法和变量。

2.6. 方法转为过程

学过程序设计实习课的同学都知道，将一个 C++类的成员函数改为 C 函数，需要将成员函数所在的对象（this 指针），变成一个新的参数。MiniJava 到 Piglet 的转换也类似，所有的方法预留一个 TEMP 0，用来存储方法所在对象。

由于不同方法中可能会有重名函数，需要加以区分。除了主类的 main 函数被强制更名为 MAIN 之外，其他函数都以类名_方法名的形式更名。由于为了防止 A 类的 B_C 方法与 A_B 类的 C 方法均被映射到同样的过程名 A_B_C，我设置了一个 HashSet 来存储所有的过程名，第二个 A_B_C 过程将会被更名为 A_B_C_2，第三个 A_B_C 更名为 A_B_C_3，依次类推。

Piglet 语言只支持传输 20 个参数（排除掉 TEMP 0，只支持 19 个参数），所以必须通过数组来传递多出来的参数。我的策略是，MiniJava 方法中的前 18 个参数直接映射到 TEMP 1 至 TEMP 18。第 19 个参数开始的所有参数，全部放到一个数组中，这个数组的指针作为第 19 个参数（即 Piglet 代码中第 20 个参数）传给过程，然后过程再用 HLOAD 语句读取多余的参数即可。

以上三个转换均可在用 visitor 扫描代码前的预处理工作中完成。

2.7. 代码翻译

与类型检查类似，我定义了一个 Convert2PigletVisitor 来执行代码翻译工作。同时，我使用了 MPiglet 类保存翻译出来的 Piglet 代码，这个类的定义如下：

MPiglet 类成员变量：

类型	变量名	功能
String	szCode	保存中间代码
int	nDigit	若这段代码能翻译为一个 IntegerLiteral，这个整数的值
int	nTemp	若这段代码能翻译为一个 Temp，临时单元的编号

MPiglet 类成员方法：

类型	方法名	参数	功能
N/A	MPiglet	szName:如果 Piglet 代码翻译自标识符，此为名称 szType:如果 Piglet 翻译自表达式，此为类型 szCode:Piglet 代码	构造函数
void	check	N/A	检查 Piglet 代码是否可以翻译成一个 Temp 或者一个 IntegerLiteral
int	getTemp	N/A	假如是一个 Temp，返回 Temp 编号
boolean	isTemp	N/A	返回是否是一个 Temp
boolean	isDigit	N/A	返回是否是一个 IntegerLiteral
String	toString	N/A	输出代码
void	append	szNewCode：新代码	将新代码加到自己代码之后
void	append	newPiglet：新代码的 MPiglet	将新代码加到自己代码之后
void	setType	szType:设置类型	如果该 MPiglet 是一个 Expression 翻译而来的
void	format	N/A	格式化，调整缩进

翻译的时候，针对每一条产生式，产生一个 MPiglet 对象，这个 MPiglet 对象是在访问了产生式右端各非终结符号/终结符号并返回了 MPiglet 对象后，对这些 MPiglet 对象综合而产生的。

在这里我做了一点小小的优化，对于如数组，对象等表达式，如果它们翻译出来的代码已经是 TEMP 就不作处理，如果不是 TEMP 的话我会先把它们通过 MOVE 指令变成一个 TEMP，这样也减少了以后翻译的工作量。

在翻译中一些需要注意的细节：1.true, false 被 0,1 代替，! x 被 1-x(即 MINUS 1 TEMP X)代替，this 被 TEMP 0 代替，成员变量由于保存在数组中需要通过 HLOAD/HSTORE 读写，而局部变量可以直接通过 TEMP 单元访问。

下面给出几个较为重要的实例。

2.8. 创建数组/对象

假如要创建一个数组 A ,长度为 L ,需要开辟 4L+4 大小的内存空间 ,然后偏移位置为 0 的地方存入 L，之后所有元素全部初始化为 0.在这一步，我检查了设置的数组长度是否大于 0.

```
    Public MPiglet visit(ArrayAllocationExpression n, Mtype argu)//创建数组
    {
        n.f0.accept(this, argu);
        n.f1.accept(this, argu);
        n.f2.accept(this, argu);

        int nArrayTemp, nSizeTemp, nLoopTemp, nRearTemp, nLabel1, nLabel2, nLabel3, nLabel4;
        String szLengthExp;

        nLabel1 = nCurrentLabel++;
        nLabel2 = nCurrentLabel++;
        nLabel3 = nCurrentLabel++;
        nLabel4 = nCurrentLabel++;

        MPiglet ret = new MPiglet(null, null, "\nBEGIN\n");
        MPiglet exp = n.f3.accept(this, argu);
        if (exp.isDigit() || exp.isTemp())//数组长度是立即数或者TEMP
        {
            szLengthExp = exp.toString();
        }
        else//载入数组长度
        {
            int nLengthTemp = nCurrentTemp++;

            ret.append("MOVE TEMP " + nLengthTemp + " ");
            ret.append(exp);
            ret.append("\n");

            szLengthExp = "TEMP " + nLengthTemp;
        }

        ret.append("CJUMP LT 0 " + szLengthExp + " L" + nLabel3 + "\n");
        //检查数组长度是否正确
        nSizeTemp = nCurrentTemp++;
        ret.append("MOVE TEMP " + nSizeTemp + " TIMES 4 PLUS 1 "+ szLengthExp + "\n");
        //数组占内存大小

        nArrayTemp = nCurrentTemp++;
        ret.append("MOVE TEMP " + nArrayTemp + " HALLOCATE TEMP " + nSizeTemp + "\n");
        //分配内存
        ret.append("HSTORE TEMP " + nArrayTemp + " 0 " + szLengthExp + "\n");//存入数组长度

        nRearTemp = nCurrentTemp++;
        ret.append("MOVE TEMP " + nRearTemp + " PLUS TEMP " + nArrayTemp + " TEMP " + nSizeTemp
+ "\n");//数组末端位置

        nLoopTemp = nCurrentTemp++;
        ret.append("MOVE TEMP " + nLoopTemp + " PLUS 4 TEMP " + nArrayTemp + "\n");
        //从第一个元素开始循环
        ret.append("L" + nLabel1 + "\tCJUMP LT TEMP " + nLoopTemp + " TEMP " + nRearTemp + " L"
+ nLabel2 + "\n");//到结束为止
        ret.append("HSTORE TEMP " + nLoopTemp + " 0 0\n");//每个元素均设为0
        ret.append("MOVE TEMP " + nLoopTemp + " PLUS 4 TEMP " + nLoopTemp + "\n");
        //下一个元素
        ret.append("JUMP L" + nLabel1 + "\n");//继续循环

        ret.append("L" + nLabel2 + "\tJUMP L" + nLabel4 + "\n");
        ret.append("L" + nLabel3 + "\tERROR\n");//数组为NULL或越界，报错
        ret.append("L" + nLabel4 + "\tNOOP\n");

        ret.append("RETURN TEMP " + nArrayTemp + "\n");//返回数组
        ret.append("END");

        n.f4.accept(this, argu);
        return ret;
    }
```

举例：创建一个长度为 10 的数组（new int[10];）

```
BEGIN
     CJUMP LT 0 10 L4
//如果数组长度≤0，则跳转到L4(虽然这里是常数10不需比较，但是实际操作中可能会是TEMP, Exp或0)
     MOVE TEMP 27 TIMES 4 PLUS 1 10        //数组占据空间大小为4*(10+1)
     MOVE TEMP 28 HALLOCATE TEMP 27        //分配内存
     HSTORE TEMP 28 0 10                   //在偏移地址为0处存入长度
     MOVE TEMP 29 PLUS TEMP 28 TEMP 27     //计算尾地址（即最后一个元素地址后的那个地址）
     MOVE TEMP 30 PLUS 4 TEMP 28           //将第一个元素的地址放入TEMP30
L2   CJUMP LT TEMP 30 TEMP 29 L3           //如果已经达到了尾地址，结束循环
     HSTORE TEMP 30 0 0                    //将当前元素初始化为0
     MOVE TEMP 30 PLUS 4 TEMP 30           //偏移地址加四，开始初始化下一个约束
     JUMP L2                               //继续循环
L3   JUMP L5
L4   ERROR                                 //数组长度为0或者负数，报错退出
L5   NOOP
     RETURN TEMP 28                        //返回数组的基地址
END
```

创建对象与数组类似，不同的是，需要创建两个数组，长度分别是 4*（本类和所有基类中成员变量个数+1）和 4*本类和所有基类中不互相覆盖的成员函数个数，所有的成员变量同样要设初值为 0。由于和创建数组类似，不再贴出代码。

2.9. 逻辑与表达式

C 语言中的逻辑与又被称为短路与，对于表达式 A&&B，程序运行时若 A 的值为假，则不会再检查 B 的值。Java 语言同样是这样，特别是对于表达式 A&&B.c()，由于 B.c()是一个方法，它的调用与否可能会导致数据的改变，所以逻辑与不能简单地处理成一个乘法，操作符左边和右边的表达式必须分别处理：

```
     int nLabel1, nLabel2, nTemp;

     nLabel1 = nCurrentLabel++;
     nLabel2 = nCurrentLabel++;
     nTemp = nCurrentTemp++;//要返回的值
     MPiglet ret = new MPiglet(null, null, "\nBEGIN\n");
     ret.append("CJUMP ");
     ret.append(n.f0.accept(this, argu));
     ret.append(" L" + nLabel1 + "\n");
     //第一个表达式为0则不再进入第二个表达式的计算（同标准JAVA）
     n.f1.accept(this, argu);
     ret.append("CJUMP ");
     ret.append(n.f2.accept(this, argu));
     ret.append(" L" + nLabel1 + "\n");
     ret.append("MOVE TEMP " + nTemp + " 1\n");
     ret.append("JUMP L" + nLabel2 + "\n");
     ret.append("L" + nLabel1 + "\tMOVE TEMP " + nTemp + " 0\n");
     ret.append("L" + nLabel2 + "\tNOOP\n");

     ret.append("RETURN TEMP " + nTemp + "\n");
     ret.append("END");
```

举例：表达式 a&&b 将被翻译成：

```
BEGIN
     CJUMP TEMP 22 L4   //a == false则跳至L4
     CJUMP TEMP 21 L4   //b == false则跳至L4
     MOVE TEMP 29 1     //a && b = true
     JUMP L5
L4   MOVE TEMP 29 0     //a && b = false
L5   NOOP
     RETURN TEMP 29     //返回a && b
END
```

2.10. 方法调用

通过 MessageSend 调用方法时，如调用 a.b()，假设对象 a 已在临时单元 TEMP 21 中，通过 HLOAD TEMP 22 TEMP 21 0 获取虚拟函数表，然后通过 HLOAD TEMP 23 TEMP 22 0(假设 b 是创建对象时存入的第一个方法)，这时 TEMP 23 即为所要调用的过程，通过 CALL 语句调用即可。

这里我也做了对象空引用指针的检查，防止被调用方法的对象为 NULL。此外，如果调用的参数超过 18 个（加上对象本身为 19 个参数），那么从第 19 个参数开始的所有参数先存到一个新开辟的数组中，然后再将这个数组的引用作为第 19 个参数（Piglet 代码中第 20 个参数）传给被调用的过程。

```
    Public MPiglet visit(MessageSend n, Mtype argu)//方法调用
    {
        int nLabel1, nLabel2, nMethodListTemp, nMethodTemp, nObjectTemp;

        nLabel1 = nCurrentLabel++;
        nLabel2 = nCurrentLabel++;

        MPiglet ret = new MPiglet(null, null, "CALL\nBEGIN\n");
        MPiglet exp = n.f0.accept(this, argu);//获取表达式
        if (exp.isTemp())//获取表达式
        {
            nObjectTemp = exp.getTemp();
        }
        else {
            nObjectTemp = nCurrentTemp++;
            ret.append("MOVE TEMP " + nObjectTemp + " ");
            ret.append(exp);
            ret.append("\n");
        }
        ret.append("CJUMP LT 0 TEMP " + nObjectTemp + " L" + nLabel1 + "\n");//对象是否为空

        nMethodListTemp = nCurrentTemp++;//方法表
        ret.append("HLOAD TEMP " + nMethodListTemp + " TEMP " + nObjectTemp + " 0\n");
        //载入方法表
        n.f1.accept(this, argu);
        String szClass = exp.getType();
        MClass _class = classList.get(szClass);//获取类
        String szMethod = n.f2.accept(this, argu).getName();
        MMethod _method = _class.getMethod(szMethod);//获取方法
        String szReturnType = _method.getType();
        MClass returnClass = classList.get(szReturnType);
        if (!returnClass.isBasicType()) ret.setType(szReturnType);//设置返回类型
        nMethodTemp = nCurrentTemp++;//方法
        ret.append("HLOAD TEMP " + nMethodTemp + " TEMP " + nMethodListTemp + " " +
_method.getOffset() + "\n");//偏移位置
        n.f3.accept(this, argu);
        MactualParamList paramList = new MactualParamList(n.f3.beginLine, (MIdentifier)
argu);//实参列表
        n.f4.accept(this, paramList);
        Vector<MPiglet> pigletList = paramList.getPiglets();//参数的中间代码表
        int nParamList = pigletList.size();//参数个数
        boolean bExtraParam = nParamList > 18;//参数个数是否超出
        String preEnd = "", postEnd = "";
        if (bExtraParam) {
            int nExtraParamTemp = nCurrentTemp++;//额外的参数
            preEnd += "MOVE TEMP " + nExtraParamTemp + " HALLOCATE " +
_method.getExtraParamSize() + "\n";
            for (int i = 18; i < nParamList; ++i) {
                preEnd += "HSTORE TEMP " + nExtraParamTemp + " " + (4 * (i - 18)) + " ";
                //间接存储
                preEnd += pigletList.get(i);
                preEnd += "\n";
            }
            for (int i = 0; i < 18; ++i)//加入各参数
            {
                postEnd += " ";
                postEnd += pigletList.get(i);
            }
```

```
            postEnd += " TEMP " + nExtraParamTemp;//额外参数
        }
        else {
            for (int i = 0; i < nParamList; ++i)//加入各参数
            {
                postEnd += " ";
                postEnd += pigletList.get(i);
            }
        }
        ret.append(preEnd);
        ret.append("JUMP L" + nLabel2 + "\n");
        ret.append("L" + nLabel1 + "\tERROR\n");
        ret.append("L" + nLabel2 + "\tNOOP\n");
        ret.append("RETURN TEMP " + nMethodTemp + "\n");
        ret.append("END (TEMP " + nObjectTemp);//二次间接地址
        ret.append(postEnd);
        ret.append(")");
        n.f5.accept(this, argu);
        return ret;
    }
    public MPiglet visit(ExpressionList n, Mtype argu)//实参列表
    {
        ((MactualParamList) argu).addPiglet(n.f0.accept(this, argu));//加入中间代码
        n.f1.accept(this, argu);
        return null;
    }
    public MPiglet visit(ExpressionRest n, Mtype argu)//实参
    {
        n.f0.accept(this, argu);
        ((MactualParamList) argu).addPiglet(n.f1.accept(this, argu));//加入中间代码
        return null;
    }
```

举例：Online IDE Testcase 2 中

表达式 new BT().f(1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19, 20, 21, 22, 23, 24, 25) 被翻译成了：

```
CALL
    BEGIN
        MOVE TEMP 25
        BEGIN
            MOVE TEMP 23 HALLOCATE 4    //给BT类的虚拟函数表分配4字节
            MOVE TEMP 24 HALLOCATE 4    //给BT类对象分配4字节
            HSTORE TEMP 23 0 BT_f       //将BT.f存入虚拟函数表
            HSTORE TEMP 24 0 TEMP 23    //将虚拟函数表存入BT类对象
            RETURN TEMP 24              //返回BT类对象
        END
        CJUMP LT 0 TEMP 25 L0           //检查BT类对象是否为空指针
        HLOAD TEMP 26 TEMP 25 0         //载入虚拟函数表
        HLOAD TEMP 27 TEMP 26 0         //载入BT.f函数翻译得到的过程
        MOVE TEMP 28 HALLOCATE 28       //为多余的7个参数分配一个数组
        HSTORE TEMP 28 0 19             //依次存入7个参数
        HSTORE TEMP 28 4 20
        HSTORE TEMP 28 8 21
        HSTORE TEMP 28 12 22
        HSTORE TEMP 28 16 23
        HSTORE TEMP 28 20 24
        HSTORE TEMP 28 24 25
        JUMP L1
    L0  ERROR                           //对象空引用报错
    L1  NOOP
        RETURN TEMP 27                  //返回过程
    END (TEMP 25 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 TEMP 28)
    //调用过程，其中第1个参数为BT类对象本身，第20个参数为原MiniJava方法额外参数的数组。
```

2.11. 安全性检查

为了保证数据安全，避免访问到不该访问的数据（如数组越界），以及未初始化的数据（如未初始化的局部变量，数组/对象空引用），采取了如下措施：

1) 所有的局部变量在其定义时就初始化为 0；

2) 所有对对象和数组的引用都必须检查是否非空（即不为 0），否则跳转到 ERROR

3) 对于数组元素的访问，下标必须在[0，L-1]的闭区间中（L 为数组长度），否则跳转到 ERROR.

以访问数组元素为例：

```
    public MPiglet visit(ArrayLookup n, Mtype argu) {//取数组元素
        int nLabel1, nLabel2, nArrayTemp, nLengthTemp, nElementTemp;
        String szIndexExp;
        nLabel1 = nCurrentLabel++;
        nLabel2 = nCurrentLabel++;
        MPiglet ret = new MPiglet(null, null, "\nBEGIN\n");
        MPiglet exp1 = n.f0.accept(this, argu);//获取数组表达式
        if (exp1.isTemp()) {//是TEMP（不可能是立即数）
            nArrayTemp = exp1.getTemp();
        }
        else {//不是TEMP，载入
            nArrayTemp = nCurrentTemp++;
            ret.append("MOVE TEMP " + nArrayTemp + " ");
            ret.append(exp1);
            ret.append("\n");
        }
        ret.append("CJUMP LT 0 TEMP " + nArrayTemp + " L" + nLabel1 + "\n");//数组是否为NULL
        nLengthTemp = nCurrentTemp++;
        ret.append("HLOAD TEMP " + nLengthTemp + " TEMP " + nArrayTemp + " 0\n");
        //载入数组长度
        n.f1.accept(this, argu);
        MPiglet exp2 = n.f2.accept(this, argu);
        if (exp2.isDigit() || exp2.isTemp()) {//下标是数字或TEMP
            szIndexExp = exp2.toString();
        }
        else {//载入下标
            int nIndexTemp = nCurrentTemp++;

            ret.append("MOVE TEMP " + nIndexTemp + " ");
            ret.append(exp2);
            ret.append("\n");

            szIndexExp = "TEMP " + nIndexTemp;
        }
        ret.append("CJUMP LT 0 PLUS 1 " + szIndexExp + " L" + nLabel1 + "\n");
        //检查数组是否下越界
        ret.append("CJUMP LT " + szIndexExp + " TEMP " + nLengthTemp + " L" + nLabel1 + "\n");
        //检查数组是否上越界
        nElementTemp = nCurrentTemp++;
        ret.append("HLOAD TEMP " + nElementTemp + " PLUS TEMP " + nArrayTemp + " TIMES 4 PLUS
1 " + szIndexExp + " 0\n");
        n.f3.accept(this, argu);
        ret.append("JUMP L" + nLabel2 + "\n");
        ret.append("L" + nLabel1 + "\tERROR\n");//数组越界或为NULL
        ret.append("L" + nLabel2 + "\tNOOP\n");
        ret.append("RETURN TEMP " + nElementTemp + "\n");
        ret.append("END");
        return ret;
    }
```

在上例中，访问数组元素前，我先检查数组是否为 NULL，是则报错，不是再检查是否下标越界。需要注意，虽然 BNF 范式决定了 MiniJava 不能直接输入一个负数，但是我们仍然能够通过如 0-1(即 MINUS 0 1) 这种表达式得到一个负数。通过 PgInterpreter 已经证实了这一点，所以数组不仅要检查上越界，也要检查下越界。

3. Piglet → SPiglet

3.1. 任务要求

本步的主要目的是把表达式较为复杂的 Piglet 语言转换为更接近三地址代码的 SPiglet 语言，需要消除嵌套表达式，并且引入了简单表达式概念，绝大部分语句对所使用的表达式有了限制。由于 Piglet 与 SPiglet 高度近似，本步骤相对来说简单一些。

难度系数：2

3.2. SPiglet 语言简介

SPiglet 与 Piglet 非常接近，主要区别包括：

1) 没有“嵌套表达式”

2) 语句中，只有 move 可以使用 表达式，print 可以使用简单表达式，其他语句均用临时变量，为后续翻译提供方便。(临时变量与寄存器对应)

3) 表达式只有 简单表达式 (SimpleEXP :临时变量、整数、标号) ,调用 (Call) ,内存分配 (HAllocate) 二元运算 (BinOp)四种。

BNF 范式与 Piglet 不同的有：

CJumpStmt ::= **“CJUMP”** ***Temp*** **Label**

HStoreStmt ::= **“HSTORE”** ***Temp*** **IntegerLiteral** ***Temp***

HLoadStmt ::= **“HLOAD” Temp** ***Temp*** **IntegerLiteral**

PrintStmt ::= **“PRINT”** ***SimpleEXP***

Exp ::= ***SimpleEXP*** **| Call | HAllocate | BinOp**

StmtExp ::= **“BEGIN” StmtList “RETURN”** ***SimpleEXP*** **“END”**

Call ::= **“CALL”** ***SimpleEXP*** **“(“ (** ***Temp*** **)* “)”**

HAllocate ::= **“HALLOCATE”** ***SimpleEXP***

BinOp ::= **Operator** ***Temp SimpleEXP***

SimpleExp ::= ***Temp*** **|** ***IntegerLiteral*** **|** ***Label***

3.3. 任务分析

第一步，扫描整个 Piglet 程序，统计所有已经用过的临时单元序号。因为此步转换势必会增添更多的临时单元，所以先记下已使用的最大临时单元编号 N，之后创建临时单元的编号自 N+1 始。

第二步，再次扫描此程序，在所有出现 Exp 的语句，判断这个 Exp 是不是 SimpleExp/TEMP，如果 SPiglet 中此语句要求该表达式为 SimpleExp 或 TEMP，而这个表达式不是，则在前面增加一条 MOVE 语句把表达式转移到一个新增的临时单元。再把原语句中的表达式换为该临时单元即可。

作业通过 CountTempVisitor 和 Convert2SPigletVisitor 实现了这两步功能，另外创建了 Pexp 类来保存表达式，记录表达式是否为简单表达式/临时单元。

由于此步转换较为简单，仅给出 PExp 类的定义和一个转换函数：

PExp 类成员变量：

类型	变量名	功能
String	szExp	表达式的代码
boolean	bSimple	表达式是否是简单表达式
boolean	bTemp	表达式是否是临时单元

PExp 类成员方法：

类型	方法名	参数	功能
N/A	PExp	szExp：Piglet 中表达式代码	根据 Piglet 代码产生表达式
N/A	PExp	szExp：Piglet 中表达式代码 nFlag：SimleExp/Temp 标志	根据 Piglet 代码产生表达式 若 nFlag == 1，表达式为简单表达式 若 nFlag == 3，既是临时单元也是简单表达式
String	toString	N/A	输出表达式代码
boolean	isTemp	N/A	返回是否是一个 Temp
boolean	isSimple	N/A	返回是否是一个 SimpleExp
void	append	szNewExp：新表达式代码	将新表达式代码加到自己代码之后
void	append	newExp：新表达式的 PExp	将新表达式代码加到自己代码之后

下面给出一个 Piglet 到 SPiglet 的转换函数，以 CJumpStmt 为例：

在 Piglet 中产生式为： **CJumpStmt::="CJUMP" Exp Label**

在 SPiglet 中产生式为： **CJumpStmt::="CJUMP"** ***Temp*** **Label**

```
public PExp visit(CJumpStmt n, Node argu)//条件跳转
{
     PExp exp1 = n.f1.accept(this, n);
     PExp exp2 = n.f2.accept(this, n);
     if (!exp1.isTemp())//表达式1必须是TEMP
     {
          int nTemp = nCurrentTemp++;
          println("MOVE TEMP " + nTemp + " " + exp1);//移入TEMP
          exp1 = new PExp("TEMP " + nTemp, PExp.TEMP);//替换成TEMP
     }
     println("CJUMP " + exp1 + " " + exp2);
     return null;
}
```

上为 CJumpStmt 转换函数。如 if (a && b) … else …翻译成 Piglet 和 SPiglet 代码分别为：

```
//Piglet代码
CJUMP              //CJump后是一个StmtExp
     BEGIN
          CJUMP TEMP 21 L4
          CJUMP TEMP 20 L4
          MOVE TEMP 28 1
          JUMP L5
     L4   MOVE TEMP 28 0
     L5   NOOP
          RETURN TEMP 28
     END L2
```

```
//SPiglet代码
     CJUMP TEMP 21 L4
     CJUMP TEMP 20 L4
     MOVE TEMP 28 1
     JUMP L5
L4   MOVE TEMP 28 0
L5   NOOP
     CJUMP TEMP 28 L2
//原来CJump后的Exp被移到一个Temp中
```

3.4. 其他

注意，所有的表达式转换函数都应该按照后序遍历的顺序，先转换所有子表达式，将子表达式转换多出来的语句放在前面，再转换父表达式。

为控制 SPiglet 代码缩进，Piglet 代码中所有语句((**Label**)? **Stmt**)前的 Label 都加上了字符串“----”，格式化时再替换掉这个字符串，并且减少一格制表符缩进。

4. SPiglet → Kanga

4.1. 任务要求

本步要求把可以使用无限个临时单元的 SPiglet 语言转换为只能使用有限个寄存器的 Kanga 语言，需要对所有临时单元使用的活性进行分析，通过临时单元在不同基本块/语句中的活性情况，分配寄存器给临时单元。在此步，还可以对输入的代码进行流图分析，进行一定的优化。

重点：划分基本块、流图分析、活性分析、寄存器分配、优化代码

难度系数：5

4.2. Kanga 语言简介

Kanga 是面向 MIPS 的语言，与 SPiglet 接近，但有如下不同：

1）标号（label）是全局的，而非局部的

2）几乎无限的临时单元变为了有限的寄存器：24 个

a) a0 - a3：存放向子函数传递的参数

b) v0 - v1：v0 存放子函数返回结果，v0、v1 还可用于表达式求值，从栈中加载

c) s0 - s7：存放局部变量，在发生函数调用时一般要保存它们的内容

d) t0 - t9：存放临时运算结果，在发生函数调用时不必保存它们的内容

3）开始使用运行栈。有专门的指令用于从栈中加载（ALOAD）、向栈中存储（ASTORE）值

SPILLEDARG i 指示栈中的第 i 个值，第一个值在 SPILLEDARG 0 中

4）Call 指令的格式发生较大的变化

a) 没有显式调用参数，需要通过寄存器传递

b) 没有显式“RETURN”

c) a0-a3 存放向子函数传递的参数

d) 如果需要传递多于 4 个的参数，需要使用 PASSARG 指令将其它参数存到栈中，PASSARG 从 1 开始。

Kanga 与 SPiglet 语言 BNF 范式的区别有：

Procedure ::= **Label "[" IntegerLiteral "]"** *"[" IntegerLiteral "]"* *"[" IntegerLiteral "]" StmtList "END"*

Stmt ::= **NoOpStmt | ErrorStmt | CJumpStmt | JumpStmt**
|HStoreStmt |HLoadStmt | MoveStmt | PrintStmt
| ALoadStmt | AStoreStmt | PassArgStmt | CallStmt

CJumpStmt ::= **"CJUMP"** ***Reg*** **Label**

HStoreStmt ::= **"HSTORE"** ***Reg*** **IntegerLiteral** ***Reg***

HLoadStmt ::= **"HLOAD"** ***Reg Reg*** **IntegerLiteral**

MoveStmt ::= **"MOVE"** ***Reg*** **Exp**

ALoadStmt ::= ***"ALOAD" Reg SpilledArg***

AStoreStmt ::= ***"ASTORE" SpilledArg Reg***

PassArgStmt ::= ***"PASSARG" IntegerLiteral Reg***

CallStmt ::= ***"CALL" SimpleEXP***

Exp ::= **SimpleEXP | HAllocate | BinOp**

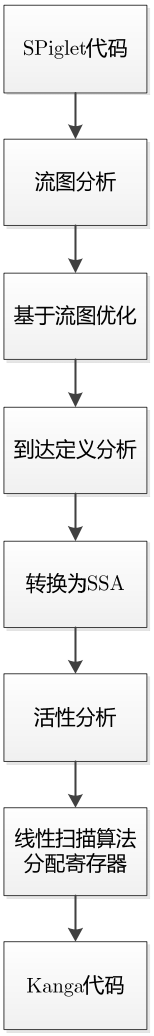
BinOp ::= **Operator** ***Reg*** **SimpleEXP**

SpilledArg **::=** ***"SPILLEDARG" IntegerLiteral***

SimpleExp **::=** ***Reg*** | **IntegerLiteral** | **Label**

所有需要使用到临时单元 Temp 的地方都被寄存器 Reg 替代，所以临时单元 Temp 和寄存器（或溢出单元）有着一一对应的关系。

4.3. 任务分析



本模块的核心任务是：临时单元活性分析，然后按照活性分析的结果进行寄存器分配，与此同时优化代码。

为了更好的分析代码，首先要把输入的代码划分成一个个基本块，进行流图分析（基本块和流图的定义请见 5.4 节）。以基本块为单位进行分析，给出每一个临时单元的活性区间，然后寄存器分配。

由于本模块作业布置时间为 11 月 15 日至 27 日，在此期间计算机系大三上的两门必修课概率统计和操作系统都进行了期中考试，为了合理安排时间，我把作业分三个时间段完成：

15 日至 17 日：划分基本块和活性分析，并且对一些不可能访问到的语句段进行删除优化；

18 日至 21 日：复习概率统计；

22 日至 23 日：实现了基本的线性扫描分配寄存器算法；

24 日至 25 日：复习操作系统；

26 日晚上及 27 日上午：加入了死代码消除优化，并且对线性扫描算法进行改进，对流图进行到达定义分析，引入了静态单赋值形式（Static single assignment form，SSA，Java 6 开始即采用此方法）来改写临时单元，提高了寄存器的复用率。

但是遗憾的是，因为忙于期中考试以及其他课的内容，我没有来得及对代码进行更多的优化，如可用表达式分析，循环不变量外提，消除复制传播、常量传播均未实现。

本作业使用了 CreateFlowGraphVisitor，ReachingDefVisitor，LivenessAnalyzerVisitor，RegisterAllocatorVisitor 四个 Visitor 扫描了 SPiglet 代码四遍，并且使用了 SMethod,SBlock,SStatement 三个类表示过程→基本块→语句的关系，另有 STemp 类维护临时单元。

4.4. 基本块与流图

基本块(Basic Block)：

1) 程序代码语句的集合；
2) 控制流只能从第一个指令进入；
3) 除基本块的最后一个指令外，控制流不会跳转/停机。

流图 (Control Flow Graph)

1) 表示各基本块（可能发生的）执行次序的有向图；
2) 流图的结点代表基本块、有向边代表基本块之间的执行次序；
3) 流图可以作为优化的基础，它指出了基本块之间的控制流，可以根据流图了解到一个值是否会被使用等信息。

以 UCLA 示例 Factorial 产生的 SPiglet 中 Fac_ComputeFac 函数为例：

SPiglet 代码：

```
Fac_ComputeFac [ 2 ]
BEGIN
     MOVE TEMP 20 0
     MOVE TEMP 33 LT TEMP 1 1
     CJUMP TEMP 33 L2
     MOVE TEMP 20 1
     JUMP L3
L2   MOVE TEMP 34 0
     MOVE TEMP 35 LT TEMP 34 TEMP 0
     CJUMP TEMP 35 L4
     HLOAD TEMP 26 TEMP 0 0
     HLOAD TEMP 27 TEMP 26 0
     JUMP L5
L4   ERROR
L5   NOOP
     MOVE TEMP 36 MINUS TEMP 1 1
     MOVE TEMP 37 CALL TEMP 27 (TEMP 0 TEMP 36)
     MOVE TEMP 20 TIMES TEMP 1 TEMP 37
L3   NOOP
     RETURN TEMP 20
END
```

划分基本块后变成：

```
     MOVE TEMP 20 0
     MOVE TEMP 33 LT TEMP 1 1
     CJUMP TEMP 33 L2
```

```
     MOVE TEMP 20 1
     JUMP L3
```

```
L2   MOVE TEMP 34 0
     MOVE TEMP 35 LT TEMP 34 TEMP 0
     CJUMP TEMP 35 L4
```

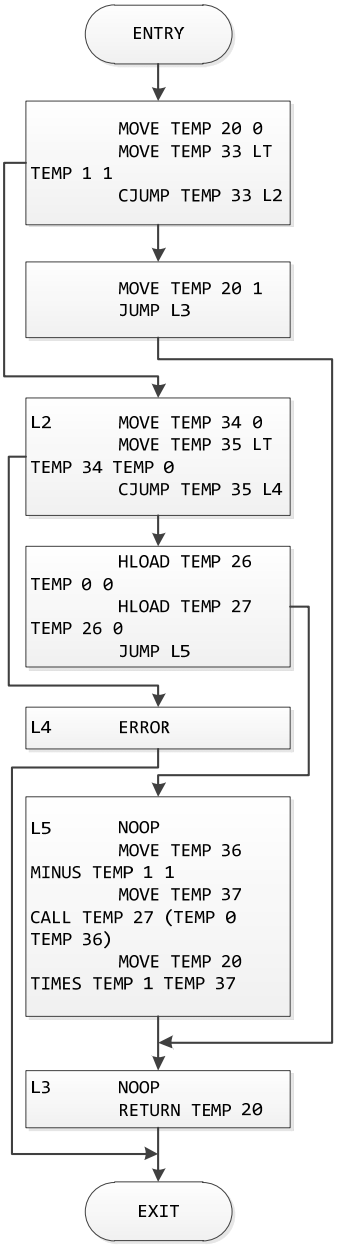
```
     HLOAD TEMP 26 TEMP 0 0
     HLOAD TEMP 27 TEMP 26 0
     JUMP L5
```

```
L4   ERROR
```

```
L5   NOOP
     MOVE TEMP 36 MINUS TEMP 1 1
     MOVE TEMP 37 CALL TEMP 27 (TEMP 0 TEMP 36)
     MOVE TEMP 20 TIMES TEMP 1 TEMP 37
```

```
L3   NOOP
     RETURN TEMP 20
```

分成了 7 个基本块，可以建立如下流图：



注：

在实际转换时，Piglet 代码的 CALL 表达式在创建流图的时候被我预分解为一连串的 MOVE 语句和 CALL 语句(如果参数多于四个还有 PASSARG 语句)，以便更好的进行流图分析 (特别是死代码消除时必须先做好此步处理，原因会在后面解释)。由 MOVE CALL 表达式转换而来的语句必然处于同一个基本块中。

新增的几种特殊的产生式被我添加到了 spiglet.syntaxtree.special 包中。

此外 ,由于 PgInterpreter 中执行到 ERROR 语句后会立即输出错误并终止程序执行，所以我认为 ERROR 语句所在的基本块会直接指向出口。所以过程中指向出口的基本块不唯一，必须设置一个虚拟的基本块。而过程中入口块唯一，可以直接由第一个块代替。

4.5. 划分基本块

划分基本块的算法如下：

输入：三地址指令序列

输出：基本块的列表

方法：

1) 确定首指令（leader，基本块的第一个指令）

a) 第一个三地址指令

b) 任意一个条件或无条件转移指令的目标指令

c) 紧跟在一个条件/无条件转移指令之后的指令

2) 确定基本块：每个首指令对应于一个基本块（从首指令开始到下一个首指令）

体现在 SPiglet 代码中，在以下几种情况下可以划分为一个新块：

1) 一个过程开始时

2) CJumpStmt 或 JumpStmt 之后

3) ErrorStmt 之后

4) 句首的标签之前

按照这四个条件，Fac_ComputeFac 过程将被划分为：

```
     MOVE TEMP 20 0
     MOVE TEMP 33 LT TEMP 1 1
     CJUMP TEMP 33 L2
```

```
     MOVE TEMP 20 1
     JUMP L3
```

```
     //纯空块1
```

```
L2   MOVE TEMP 34 0
     MOVE TEMP 35 LT TEMP 34 TEMP 0
     CJUMP TEMP 35 L4
```

```
     HLOAD TEMP 26 TEMP 0 0
     HLOAD TEMP 27 TEMP 26 0
     JUMP L5
```

```
     //纯空块2
```

```
L4   ERROR
```

```
     //纯空块3
```

```
L5   NOOP
     MOVE TEMP 36 MINUS TEMP 1 1
     MOVE TEMP 37 CALL TEMP 27 (TEMP 0 TEMP 36)
     MOVE TEMP 20 TIMES TEMP 1 TEMP 37
```

```
L3   NOOP
     RETURN TEMP 20
```

这里产生了 3 个纯空块，在随后的扫描过程中，这些空块将被消除。

在上面的基本块中，有一些基本块是以标签开始的，它们可能有一个以上的前驱结点（也有可能一个也没有），没有入口标签的基本块至多有一个前驱结点。

以 Jump 语句结尾的基本块或者不以 Jump/CJump/Error 语句结尾的基本块有且只有一个后继结点，前者的后继结点由出口标签决定，后者的后继结点为基本块列表中顺序的下一个基本块。CJump 语句结尾的基本块有两个后继结点，分别是前两种情况的后继，Error 语句结尾的基本块后继结点必然是出口结点。

4.6. 构造流图

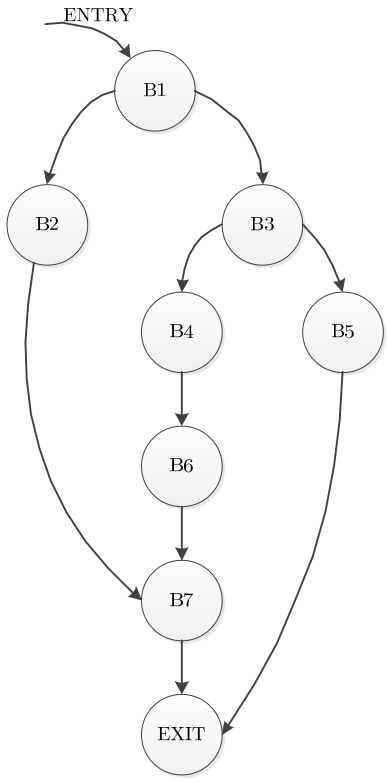
流图构造遵循如下规则：

1) 流图的顶点是基本块
2) 两个顶点 B 和 C 之间存在一条有向边 iff 基本块 C 的第一个，指令可能在 B 的最后一个指令之后执行。
3) 存在边的原因：
   a) B 的结尾指令是一条跳转到 C 的开头的条件/无条件语句
   b) 在原来的序列中，C 紧跟在 B 之后，且 B 的结尾不是无条件跳转语句（以及 Error 语句）
4) 称 B 是 C 的前驱(predecessor)，C 是 B 的后继(successor)
5) 入口(ENTRY)，出口(EXIT)结点：
   a) 流图中额外添加的边，与中间代码（基本块）不对应
   b) 入口到第一条指令有一条边
   c) 从任何可能最后执行的基本块到出口有一条边

以上节划分的基本块为例，去掉三个空块，依次编号为 B1~7，可以观察到：

1) B1 为第 1 个块，顺序执行会执行到 B2，条件跳转到 B3。无入口标签，除过程入口外无任何前驱结点；
2) B2 无入口标签，只有一个前驱结点 B1，最后一句无条件跳转到 B7，只有 B7 一个后继结点；
3) B3 只能从 B1 跳转到 L2 才能执行，只有 B1 一个前驱结点，顺序执行到 B4，条件跳转到 B5；
4) B4 无入口标签，只能从 B3 顺序执行，最后一句无条件调整到 B6；
5) B5 只能从 B3 跳转到 L4 才能执行，后继结点只有出口结点；
6) B6 只能从 B4 跳转而来，最后一句非跳转/错误指令，只能顺序执行到 B7；
7) B7 可以从 B2 跳转而来，或者从 B6 顺序执行，后继结点只有出口结点。

通过这些前驱-后继关系，我们就可以建立起一个流图：



创建流图的代码：

```
    clearEmptyBlocks();//清除空块
    int nLength = lstBlock.size();//块个数
    entryBlock = lstBlock.firstElement();//入口块
    exitBlock = new SBlock(nLength, this);//出口块
    for (int i = 0 ; i < nLength; ++i) {
        SBlock block = lstBlock.get(i);//获取当前块
        Node jumpStmt =
block.getStmts().lastElement().getNode();
        //获取最后一条指令
        if (jumpStmt instanceof ErrorStmt) continue;
        //如果报错指令，无下一条指令
        if (!(jumpStmt instanceof JumpStmt) && i <
nLength - 1) {//非无条件跳转指令，且非最后一个块
            SBlock nextBlock = lstBlock.get(i + 1);
            block.getNext().add(nextBlock);//加入后继
            nextBlock.getPrev().add(block);//加入前驱
        }
        if (block.getExitLabel() != null){//有出口标签
            SBlock nextBlock =
mapEntryLabel.get(block.getExitLabel());
            block.getNext().add(nextBlock);//加入后继
            nextBlock.getPrev().add(block);//加入前驱
        }
    }
    clearUnreferredBlocks();//清除未访问块
    clearLabels();//清除无用标签
```

4.7. 基于基本块的初步优化

前面给出的代码中没有将 B5，B7 两个基本块链接到 ExitBlock 上，这个将在清除未访问块函数中进行。在已经得到各基本块后，我们可以对程序代码进行如下优化：

1) 删除所有空块

在划分基本块时，我们可能会得到几个纯空块，直接扫描删除即可。由于纯空块不含标签和调整/错误语句，在建立流图前删除它不会影响其余结点的前驱、后继关系。

2) 删除所有未访问块：

对于如下语句：

```
     JUMP L0
     PRINT TEMP 1
L0   NOOP
```

显然，PRINT TEMP 1 语句永远不可能被访问到，所以可以把这条语句（及其所在的基本块）删除。方法是通过第一个基本块（入口块）开始，通过后继结点列表深搜（广搜也可）所有可以被访问到的结点，删除未被访问到的结点。若有结点无任何后继结点，则将出口结点添加到这个结点的后继结点。

3) 清除多余的标签：

在前两步之后，一些基本块被删除，一些出口标签也随之删除，考虑如下语句：

```
     JUMP L0
     PRINT TEMP 1
     JUMP L1
L0   NOOP
L1   NOOP
```

假设所有的跳转语句中只有 PRINT TEMP 1 后面的那个 JUMP 语句跳转到了 L1 那么随着 JUMP L1 被删除，所有依靠出口标签跳转到 L1 的基本块都已被删除。L1 标签可以被删除。

以上三种优化都很简单，所以仅给出清除未访问块的实现代码：

```
private void clearUnreferredBlocks()//清除未访问块
{
     Vector<SBlock> lstNewBlock = new Vector<SBlock>();
     DFS(lstNewBlock, entryBlock);//遍历各块
     lstBlock = lstNewBlock;
     Collections.sort(lstBlock);//排序
     for (SBlock block : lstBlock)//删除无效连接
     {
          HashSet<SBlock> lstNewPrev = new HashSet<SBlock>();
          for (SBlock prevBlock : block.getPrev())
               if (lstBlock.contains(prevBlock)) lstNewPrev.add(prevBlock);
          block.getPrev().clear();
          block.getPrev().addAll(lstNewPrev);
     }
}

private void DFS(Vector<SBlock> lstNewBlock, SBlock block)//深度优先搜索
{
     lstNewBlock.add(block);//加入此块
     if (block.getNext().isEmpty())
     {
          block.getNext().add(exitBlock);//加入出口
          exitBlock.getPrev().add(block);//加入入口
          return;
     }
     for (SBlock nextBlock : block.getNext())//遍历未访问的后继块
          if (!lstNewBlock.contains(nextBlock)) DFS(lstNewBlock, nextBlock);
}
```

4.8. 活性分析

1. 活跃变量的定义：

1) 以下条件成立时，变量 v 在语句 p 处活跃：

v 在以 p 开始的某条语句执行路径的语句 p'中被用到，且在 p 和 p'之间，变量 v 没有重新被定值。

2) 以下条件成立时，变量 v 在语句 p 处不活跃：

v 在以 p 开始直到出口结点的所有路径中都没有被用到，或者虽然在语句 p'中用到 v，但是 p 和 p'之间，变量 v 被重新定值。

活性分析可以用于寄存器分配和死代码消除。

对于寄存器分配：两个不同时处于活性区间内的变量可以分配到同一个寄存器中。

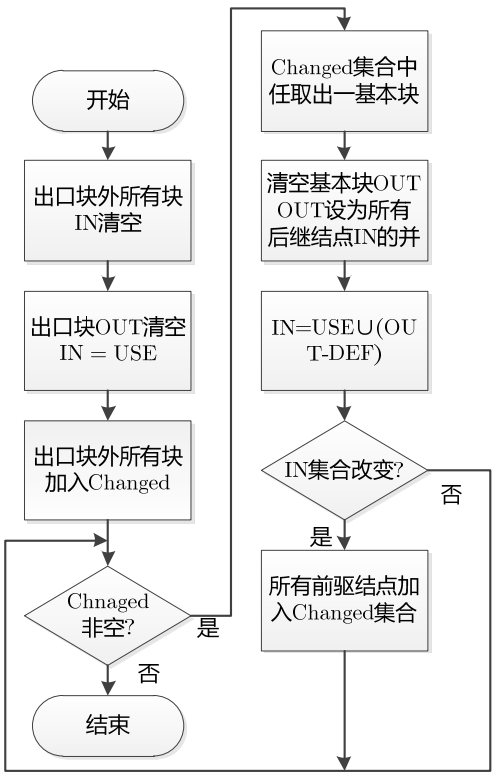
对于死代码消除：如果一个变量的定值语句之后，这个变量并没有活性，说明这个定值语句设定的值并没有被使用到，在绝大多数情况下此定值语句为死代码可以直接消除。(如果定值语句内包含了过程调用要另外考虑！)

2. 活性分析的方法：

对于每一个基本块，有以下四个集合：

集合名称	含义
IN	进基本块时活跃变量集合
OUT	出基本块时活跃变量集合
USE	基本块中使用了值（在定值前）的变量集合
DEF	基本块中定值（在使用值前）的变量集合

我编写了 LivenessAnalyzerVisitor 来统计每一个语句的 USE 和 DEF 集合(MOVE 和 HLOAD 的第一个 TEMP 参数为 DEF，其余 TEMP 为 USE)，然后汇总到每个语句所在的基本块中。获得了所有的基本块集合后，即可以进行活性分析操作。



伪代码：

```
for all nodes n in N - { Exit }
        IN[n] = emptyset;
OUT[Exit] = emptyset;
IN[Exit] = use[Exit];
Changed = N - { Exit };

while (Changed != emptyset)
        choose a node n in Changed;
        Changed = Changed - { n };

        OUT[n] = emptyset;
        for all nodes s in successors(n)
                OUT[n] = OUT[n] U IN[p];

        IN[n] = use[n] U (out[n] - def[n]);
        if (IN[n] changed)
                for all nodes p in predecessors(n)
                        Changed = Changed U { p };
```

首先将所有基本块的 IN 集合清空，然后出口块的 OUT 集合也清空，IN 集合即 USE 集合（由于我设置出口块为空块，所以 IN 集合也清空）设置 Changed 为出口块以外所有的基本块的集合。

每一次循环，若 Changed 集合未空，从集合中取出一个基本块 B，从集合中删除。清空 B 的 OUT 集合，设为其后继结点的 IN 集合的并，然后根据 IN=USE∪(OUT - DEF)算法计算出 IN 集合。如果 IN 集合发生改变，将 B 的所有前驱结点加入 Changed 集合。

重复上述循环，直到 Changed 集合为空，即到达不动点。

实现代码（分写在 SMethod，SBlock 两类中，为避免与到达定义的 IN/OUT 集合重名，活性分析的 IN/OUT 集合前面都加上了 Live 前缀）：

```
    //SMethod类的活性分析函数
    private void analyzeLiveness() {//活性分析
        for (SBlock block : lstBlock) block.initLiveness();
        HashSet<SBlock> lstChanged = new HashSet<SBlock>();
        lstChanged.addAll(lstBlock);
        while (!lstChanged.isEmpty()) {//未出现不动点
            SBlock block = lstChanged.iterator().next();//取一个基本块
            lstChanged.remove(block);
            if (block.analyzeLivenessByBlock())//如果基本块的IN集合改变
                for (SBlock prevBlock : block.getPrev()) lstChanged.add(prevBlock);
        }
        for (SBlock block : lstBlock) block.analyzeLivenessByStmt();//分析块内语句
        for (SBlock block : lstBlock) {
            HashSet<Integer> lstNewIn = new HashSet<Integer>();
            HashSet<Integer> lstOldOut = new HashSet<Integer>();
            block.removeDeadCode(lstNewIn, lstOldOut);
    }   }
    //SBlock 的活性分析函数
    protected boolean analyzeLivenessByBlock() {//按块分析
        lstLiveOut.clear();
        for (SBlock nextBlock : lstNext) lstLiveOut.addAll(nextBlock.lstLiveIn);
        HashSet<Integer> lstOldLiveIn = new HashSet<Integer>();
        lstOldLiveIn.addAll(lstLiveIn);//加入全部元素
        lstLiveIn.clear();
        lstLiveIn.addAll(lstLiveOut);
        lstLiveIn.removeAll(lstLiveDef);
        lstLiveIn.addAll(lstLiveUse);//IN=USE∪(OUT-DEF)
        return lstLiveIn.size() != lstOldLiveIn.size()
|| !lstLiveIn.containsAll(lstOldLiveIn);//返回是否改变
    }
```

据我所知，一些同学并没有划分基本块，而是直接根据每个语句建立 IN,OUT,USE,DEF 集合然后活性分析。这样做相当于把每一条语句视为一个“基本块”，在迭代计算活性变量集合时，“基本块”的个数很大，影响了程序效率。事实上，只需要确定了每一个基本块的 IN,OUT 集合后，只需要一次扫描就可以确定所有语句的 IN,OUT 集合了。

继续以 Fac_ComputeFac 过程为例：

基本块	DEF	USE	IN	OUT
B1	{20,33}	{1}	{0,1}	{0,1}
B2	{20}	∅	∅	{20}
B3	{34,35}	{0}	{0,1}	{0,1}
B4	{26,27}	{0}	{0,1}	{0,1,27}
B5	∅	∅	∅	∅
B6	{20,36,37}	{0,1,27}	{0,1,27}	{20}
B7	∅	{20}	{20}	∅

入口块需要的{0,1}即为参数。

4.9. 寄存器分配

由于 CPU 访问寄存器和访问内存时间不在一个数量级上，为了提高代码的运行效率，应尽可能多的将临时单元映射到寄存器上。

对于寄存器分配，有几种策略：

1) 只保留少数寄存器，所有临时单元全部保存在内存上，使用时再读入；

分析：这种方法浪费了几乎全部的寄存器空间，所有临时单元的访问都要通过内存，速度很差，不可采用。

2) 给所有要使用的临时单元编号，按编号给每个寄存器分配一个寄存器；

分析：这种方法稍好于前者，但是不能针对程序的实际情况进行优化，实际操作中很容易出现一些寄存器没被使用而却有临时单元被溢出到内存空间中，也不可采用。

3) 给每个临时单元设置一个活性区间，线性扫描一遍所有区间，按照区间动态分配寄存器；

分析：本次编译实习作业中被很多同学所采用的方式，简单易行，能够在一定程度上减少寄存器的使用。但是由于一个临时单元会存在多个活性区间，如果只算第一个活性区间开始到最后一个活性区间结束，加入中间无活性的时间很长，仍然会造成很大的寄存器浪费。但是这种方法可以通过静态单赋值形式等方法优化，以达到接近甚至超过图着色算法的效果。

4) 基于图着色算法分配寄存器；

分析：最为经典的算法，但是理解和实现上存在一定难度，编写和调试较花时间，且时间复杂度高，为 O(n^4)。

在本作业中采取的是第三种策略，即使用线性扫描寄存器分配算法，加以 SSA 优化，这种算法的策略是：

1) 计算每个变量的“活性区间”(live interval)

2) 维护一个“活动”(active) 队列

a) 记录放在寄存器中的变量(长度等于寄存器数目)

b) 队列中的变量以活性区间结束点为序

c) 当一个新的活性区间起始点到来时

i. 顺序扫描队列

ii. 移除过期变量

iii. 试图将新变量加入队列，当寄存器不足时，“溢出”结束最晚的变量

复杂度: O(V log R) -- V 个变量、R 个寄存器，远好图着色算法。

在作业中，线性扫描算法是在 RegisterAllocatorVisitor 中实现的，利用一个优先队列 heap 保存正在活性区间且保存在寄存器中的临时单元，堆顶的元素是活性区间结束最晚的临时单元。用集合 lstFreeReg 保存所有未被使用的寄存器。此外，寄存器 s0~s7,t0~t9 被标记为位置 0~17，溢出单元 0 被标记为位置 24，溢出单元 1 被标记为 25，依此类推。

以下是寄存器分配的代码：

```
Arrays.sort(arrTemp, new Comparator<Object>() {//所有临时单元按活性区间开始时间升序排序
    @Override
    public int compare(Object arg0, Object arg1) {
        STemp a, b;
        a = (STemp) arg0;
        b = (STemp) arg1;
        return a.getStart() - b.getStart();//比较活性区间开始时间
    }
});
for (int x : method.getTempArgs().keySet()) {//由PASSARG传的参数预留溢出单元
    if (x < 4) continue;
    int nSpilled = x - 4; //溢出单元位置，如TEMP 4被保存在SPILLEDARG 0
```

```
            lstSpilled.add(x);//溢出单元已被占用
            if (nSpilled > nSpilledSize) ++nSpilledSize;//调整使用的溢出栈大小
        }
        for (int i = 0; i < arrTemp.length; ++i)  {//所有临时单元分配寄存器/溢出单元
            STemp tmp = arrTemp[i]; //临时单元
            for (int j = 0; j < i; ++j) {
                STemp tmp1 = arrTemp[j];

                if (tmp1.getEnd() >= tmp.getStart() || tmp1.isDead()) continue;
                tmp1.die();//杀死所有活性区间已经结束且未被杀死的临时单元

                int nLocation = tmp1.getLocation();
                if (nLocation < 18) //如果被杀死的临时单元保存在寄存器中
                {
                    heap.remove(tmp1);//从堆中移除此临时单元
                    lstFreeReg.add(mapReg.get(nLocation));//增加一个可用的寄存器
                }
                else lstSpilled.remove(nLocation - 24);//增加一个可用的溢出单元
            }
            if (!lstFreeReg.isEmpty()) {//还有空闲寄存器
                Reg reg = lstFreeReg.iterator().next();//任取一个寄存器
                lstFreeReg.remove(reg);//寄存器不再空闲
                if (reg.getIndex() < 8) lstUsedSReg.add(reg);//在整个过程中已使用的S寄存器
                else lstUsedTReg.add(reg);//在整个过程中已使用的T寄存器
                tmp.setLocation(reg.getIndex());//设置保存位置
                heap.add(tmp);//堆中增加一个临时单元
            }
            else {//没有空闲寄存器
                int nSpilled, nLocation;
                for (nSpilled = 0; nSpilled < nSpilledSize && lstSpilled.contains(nSpilled);
++nSpilled);//跳过所有已占用的溢出单元
                if (nSpilled == nSpilledSize) ++nSpilledSize;//调整使用的溢出栈大小
                nLocation = nSpilled + 24;
                STemp tmp1 = heap.poll();//去除使用寄存器的结束时间最晚的临时单元
                if (tmp1.getEnd() < tmp.getEnd()){//如果被取出的临时单元结束时间早
                    tmp.setLocation(nLocation);//新临时单元放到溢出栈中
                    heap.add(tmp1);//从堆中取出的临时单元放回堆中
                }
                else {//如果新临时单元结束时间早
                    tmp.setLocation(tmp1.getLocation());//新临时单元放到旧临时单元的寄存器中
                    tmp1.setLocation(nLocation);//从堆中取出的临时单元放到溢出栈中
                    heap.add(tmp); //从新临时单元放入堆中
                }
                lstSpilled.add(nSpilled);//增加一个被使用的溢出单元
            }
        }
```

对于参数，确定了存储的寄存器/溢出单元后，需要输出一些语句，从原来的位置放到分配的位置。假如 TEMP 4 被放到了 s0 中，则应输出 ALOAD s0 SPILLEDARG 0.

上面专门保存了在整个过程中使用过的 S 寄存器和 T 寄存器，这两个寄存器在过程调用的保存时机不同。S 寄存器为 callee saved register，由被调用的过程在过程开始时，把所有将要用到的 S 寄存器备份到溢出栈中，等到过程结束时，再把被备份的 S 寄存器恢复。而 T 寄存器为 caller saved register，当一个过程要调用另一个过程时，在调用前备份所有 T 寄存器，调用后恢复。

访问溢出的临时单元，默认使用寄存器 v1，如果 v1 被另一个溢出的临时单元占用，使用寄存器 v0。由 SPiglet 产生式，表达式只有 Call 和 BinOp 会读取两个或者两个以上临时单元的值，但是 Call 已被我提前分解为一连串的 MOVE/PASSARG/CALL 操作，不会在一条语句中读取两个临时单元的值，所以只有 BinOp 表达式会需要两个溢出单元的值。

如果 MOVE TEMP 1 BinOp TEMP 2 TEMP3 三个临时单元都是被溢出的，只要先用 v0，v1 读取溢出的 TEMP 2, TEMP 3，然后将运算结果存到 v1，再通过 ASTORE 将 v1 存到 TEMP 1 的溢出单元。

4.10. 到达定义

到达定义的定义：

1) 以下条件成立，变量 v 在对 v 赋值的语句 p 中的定义能够到达语句 p'：

存在至少一条 p 开始到 p'的路径，且 p 与 p'之间没有对 v 赋值的语句。

2) 以下条件成立，变量 v 在对 v 赋值的语句 p 中的定义能够到达语句 p'：

不存在 p 开始到 p'的路径，或者所有 p 到 p'的路径中都有对 v 进行赋值的语句。

和活性分析一样，到达定义也是数据流分析中的一种，但是它是一种前向算法，通过基本块的 IN 集合推出 OUT 集合；它可以用于常量传播和复制传播，在本次作业中，到达定义被用来实现静态单赋值形式。

3. 到达分析的方法：

对于每一个基本块，有以下四个集合：

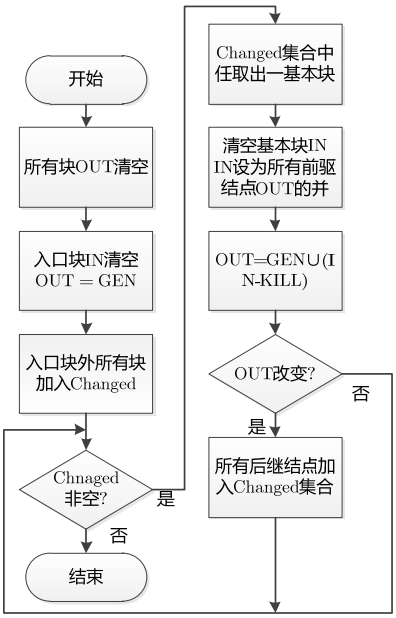
集合名称	含义
IN	进基本块时可到达的定义集合
OUT	出基本块时可到达的定义集合
GEN	基本块中产生的定义
KILL	基本块中杀死的定义

同样的，我编写了 ReachingDefVisitor 来统计每一个语句的 GEN 集合。

如果不是赋值语句，GEN 集合为∅；若这条语句是 MOVE/HLOAD，则第一个 Temp 参数即被定值的临时单元，设该语句为整个过程中第 x 条赋值语句，则这条语句所代表的定义的编号为(x+n-1)（n 为过程参数个数），且这个编号为该语句 GEN 集合中唯一元素。

扫描完代码，记录下所有定义后，再计算出每一条语句的 KILL 集合，假如某语句不是赋值语句，KILL 集合为∅，否则若被该语句赋值的临时单元为 TEMP X，那么除了这条语句以外所有对 TEMP X 赋值的定义编号都是 KILL 集合的元素。

将一个块中所有语句的 GEN/KILL 集合汇总起来，得到整个基本块的 GEN/KILL 集合。考虑一个特殊情况：如果基本块中有两条或两条以上对同一个临时单元 TEMP X 的赋值，即两个语句互相 KILL，那么出现在前面的语句 p 将会被后面的语句 q 杀死，p 的定义编号既不出现在整个基本块的 GEN 集合也不出现在 KILL 集合中，q 的定义编号出现在 GEN 集合中，不出现在 KILL 集合中。



伪代码：

```
for all nodes n in N
        OUT[n] = emptyset;
IN[Entry] = emptyset;
OUT[Entry] = GEN[Entry];
Changed = N - { Entry };
while (Changed != emptyset)
        choose a node n in Changed;
        Changed = Changed - { n };
        IN[n] = emptyset;
        for all nodes p in predecessors(n)
                IN[n] = IN[n] U OUT[p];
        OUT[n] = GEN[n] U (IN[n] - KILL[n]);
        if (OUT[n] changed)
                for all nodes s in successors(n)
                        Changed = Changed U { s };
```

与活性分析相似。首先结点的 OUT 集合置空，入口结点的 IN 集合也置空（在实际程序中是加入过程的所有参数），OUT 即 GEN 集合（实际程序中是 GEN∪(IN - KILL)）。设置 Changed 为入口块以外所有集合。（在我的程序中入口块也加入了 Changed）

每一次循环，若 Changed 集合未空，从集合中取出一个基本块 B，从集合中删除。清空 B 的 IN 集合，设为其前驱结点的 OUT 集合的并，然后根据 OUT= GEN∪(IN - KILL)法计算出 OUT 集合。如果 OUT 集合发生改变，将 B 的所有后继结点加入 Changed 集合。

重复上述循环，直到 Changed 集合为空，即到达不动点。

为避免与活性分析的 IN/OUT 集合重名，代码中到达定义的 IN/OUT 集合前面都加上了 Reach 前缀。

由于算法和活性变量类似，所以不再给出代码，继续以 Fac_ComputeFac 为例：

定值语句有：

0: TEMP 0(参数)

1: TEMP 1(参数)

2: MOVE TEMP 20 0

3: MOVE TEMP 33 LT TEMP 1 1

4: MOVE TEMP 20 1

5: MOVE TEMP 34 0

6: MOVE TEMP 35 LT TEMP 34 TEMP 0

7: HLOAD TEMP 26 TEMP 0 0

8: HLOAD TEMP 27 TEMP 26 0

9: MOVE TEMP 36 MINUS TEMP 1 1

10: MOVE TEMP 37 CALL TEMP 27 (TEMP 0 TEMP 36)

11: MOVE TEMP 20 TIMES TEMP 1 TEMP 37

基本块	GEN	KILL	IN	OUT
B1	{2,3}	{4,11}	{0,1}	{0,1,2,3}
B2	{4}	{2,11}	{0,1,2,3}	{0,1,3,4}
B3	{5,6}	∅	{0,1,2,3}	{0,1,2,3,5,6}
B4	{7,8}	∅	{0,1,2,3,5,6}	{0,1,2,3,5,6,7,8}
B5	∅	∅	{0,1,2,3,5,6}	{0,1,2,3,5,6}
B6	{9,10,11}	{2,4}	{0,1,2,3,5,6,7,8}	{0,1,3,5,6,7,8,9,10,11}
B7	∅	∅	{0,1,3,4,5,6,7,8,9,10,11}	{0,1,3,4,5,6,7,8,9,10,11}

在进行到达定义检查前，所有的参数都已经预先设置了一个定义，定义编号即参数编号本身，由于下一步静态单赋值形式会把临时单元的编号替换为其定义编号，所以参数编号不变，不影响后面的寄存器分配。

在 ReachingDefVisitor 和 LivenessAnalyzerVisitor 中，我分别各扫描了所有语句的临时单元使用情况。这看似是一种浪费，但事实上，在下一步的静态单赋值替换中，所有 Temp 变量的编号全部被改写，而且会增添进新的 MOVE 语句，原来扫描得到的 TEMP 信息已经失效。

利用到达定义，可以知道每一次使用临时单元的值时使用的是哪个定义产生的值，可以通过将临时单元的编号改为定义编号的形式，实现静态单赋值形式的转换。

与活性分析类似，基于基本块来分析到达定义后，就可以根据各基本块的 IN，OUT 集合，在线性时间内一次推出所有语句的 IN,OUT 集合，得到每一条语句使用和定义的临时单元的定义编号。

4.11. 静态单赋值形式

线性扫描寄存器算法存在很大的缺陷，考虑如下 SPiglet 语句：

```
MOVE TEMP 1 0
PRINT TEMP 1
```

若干条语句

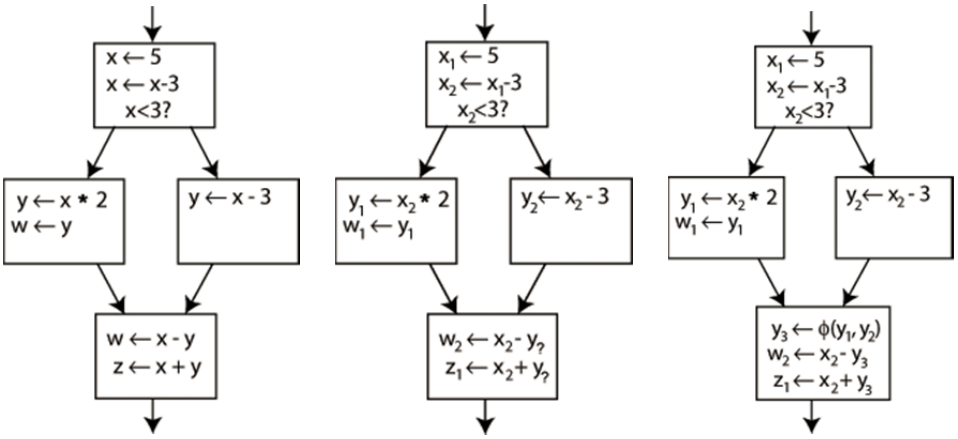
```
MOVE TEMP 1 1
PRINT TEMP 1
```

对这些语句分配寄存器，由于 TEMP 1 的第一个活性区间的开始是第一条语句，最后一个活性区间的结束是最后一条语句，那么在整个代码段中 TEMP 1 一直有活性需要占据一个寄存器。如果中间出现寄存器用完，需要溢出，如果新临时单元 TEMP 2 结束时间比 TEMP 1 晚，则 TEMP 2 被溢出到溢出栈，否则 TEMP 1 被溢出到溢出栈。如果中间这段时间实际上 TEMP 1 根本没有活性的话，就白白浪费了一个寄存器，而且还多出了读写内存的时间。

比较理想的方法就是按照 TEMP 1 的版本，分为 TEMP 1(0)和 TEMP 1(1)，两个版本分开分配寄存器，这样就需要引入静态单赋值形式。

静态单赋值（SSA）形式：一种中间代码的表示形式，在这种中间代码的过程中，每一个变量最多被赋值一次。

如下图，分别是未转换前的流程图，为变量赋予唯一名称后的流程图和加入φ结点的流程图：



单纯为每一个变量赋予唯一的名称，会造成一定的问题，如中图最下面一个基本块中 y 的取值，在不执行程序的情况下是不可能知道此处的 y 究竟是 y1 还是 y2 的，所以引入了 y3=φ(y1,y2)，分别代表两个分支的 y。

SSA 带来精确的使用-定义关系，保证每一个被使用的变量有唯一的定义，它有很多优点，对于寄存器分配，它的优点是可以把对同一个变量的不相关的若干次使用，变成对不同变量的使用，因此能消除很多不必要的依赖关系。

在作业的代码中，因为已完成到达定义的分析，每一条语句都有一个 ReachIn，ReachOut 集合，记下到达这个语句的定义，对于每一个基本块，如果有一条语句使用临时单元 TEMP X，在其 ReachIn 集合中有两条或者两条以上的定义，那么说明有多个分支产生了对 TEMP X 不同的赋值，且都到达了这条语句。这时，对基本块插入一个φ结点，存入同一个临时单元的各定义编号，并且选择一个作为被采用的编号。然后再改写语句中的定义和使用的临时单元编号为其定义编号。

此步代码（SMethod. reachingDefinitions 函数中，与语句的到达定义计算合在一起）

```
HashSet<Integer> lstPrevOut = block.getReachIn();
for (SStatement stmt : block.getStmts())
{
    HashSet<Integer> lstReachIn = stmt.getReachIn();
    HashSet<Integer> lstReachOut = stmt.getReachOut();
    lstReachIn.clear();
    lstReachIn.addAll(lstPrevOut);//先根据基本块的IN/OUT计算到达定义
    for (Temp tmp : stmt.getUseTemp()) {//所有使用值的Temp
```

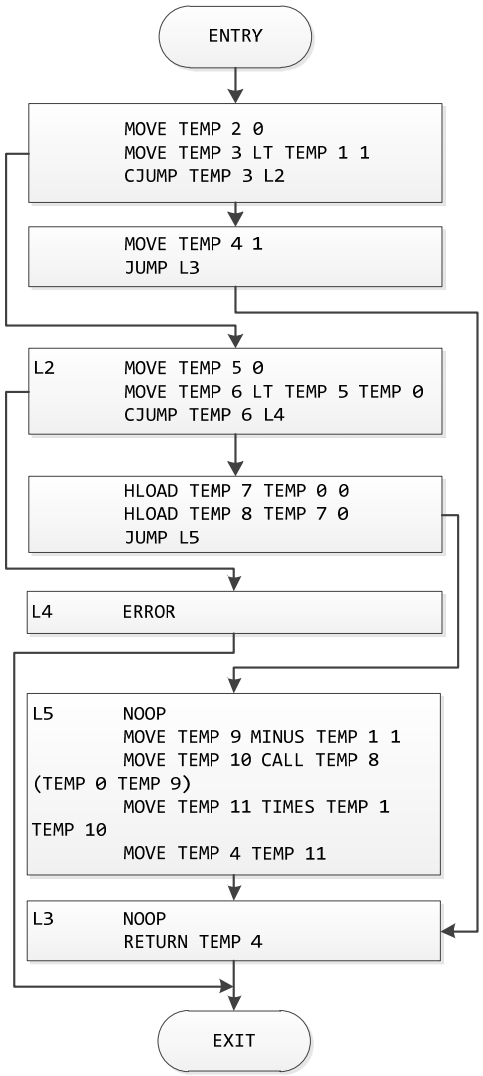
```
                int nTemp = Integer.parseInt(tmp.f1.f0.tokenImage);
                for (int x : lstReachIn) if (nTemp == mapDef.get(x)) tmp.lstVersion.add(x);
                //对于使用了值的TEMP X，记下到达该语句的X的定义
                int nVersion;
                if (tmp.lstVersion.contains(nTemp)) nVersion = nTemp;
                //如果定义序号有和临时单元相等的，优先使用（为了保证参数序号不变）
                else nVersion = tmp.lstVersion.iterator().next();
                //任选一个定义序号为新临时单元编号
                if (tmp.lstVersion.size() > 1) block.addPhi(nVersion, tmp.lstVersion);
                //超过一个定义，添加φ结点
                tmp.f1.f0.tokenImage = Integer.toString(nVersion);//改写编号
            }
            lstReachOut.clear();
            lstReachOut.addAll(lstReachIn);
            lstReachOut.removeAll(stmt.getKill());//OUT = IN - KILL
            int nGen = stmt.getGen();
            if (nGen >= 0) {
                lstReachOut.add(nGen);//OUT = GEN ∪ {IN – KILL}
                Temp defTmp = stmt.getDefTemp();//获取定义的TEMP
                defTmp.f1.f0.tokenImage = Integer.toString(nGen);//编号改写成定义序号
            }
            lstPrevOut = lstReachOut;
```

然后对于所有φ结点，除了被采用的那个定义序号外，所有其他定义序号对应的块最后插入一条 MOVE 指令，如被采用的序号为 X，另外一个序号为 Y，它们都是对同一个临时单元的赋值。则在 Y 定义所在的块最后加上 :MOVE TEMP X TEMP Y。（这里新增的 TEMP 编号不会再被上述函数改写）

SMethod. reachingDefinitions 函数中，对每一个φ结点所在的块最后添加 MOVE 语句：

```
    Hashtable<Integer, HashSet<Integer>> mapPhi =
block.getPhi();//获取φ结点
    for (int x : mapPhi.keySet()) {
        for (int y : mapPhi.get(x)) {
            if (x == y) continue;
            mapDefBlock.get(y).addNewVersion(x,
y); //对定义y所在的块最后添加MOVE语句
        }
    }
```

这里说的块最后，如果块是以 JUMP / CJUMP / ERROR / RETURN 结尾的话，插入到倒数第二句后，否则插入到倒数第一句后。右图为更改了版本后的 Fac_ComputeFac 过程流图。



临时单元被重新命名，原 TEMP 20 被分别重命名为 TEMP 2， TEMP 4， TEMP 11，在 B7 选择使用 TEMP 4.

基本块 B6 的最后加上一行 MOVE TEMP 4 TEMP 11, 而基本块 B1 的 TEMP 2 不会到达 B7,所以不用添加 MOVE TEMP 4 TEMP 2.

这里一些临时单元被重命名为 TEMP 2~11，但是由于实现已经保存参数个数（2），TEMP 2~11 不会当做参数处理。

所有的参数在第一个活性区间内编号不变，以保证寄存器分配过程中对参数进行预处理（从 a 寄存器或溢出单元中取出并放到分配算法指定的位置）。它被重新赋值后会被重命名为另外的编号。

4.12. 死代码消除

继续考虑 Fac_ComputeFac 过程，经过静态单赋值形式转换后

```
Fac_ComputeFac [ 2 ]
BEGIN
     MOVE TEMP 2 0
     MOVE TEMP 3 LT TEMP 1 1
     CJUMP TEMP 3 L2
     MOVE TEMP 4 1
     JUMP L3
L2   MOVE TEMP 5 0
     MOVE TEMP 6 LT TEMP 5 TEMP 0
     CJUMP TEMP 6 L4
     HLOAD TEMP 7 TEMP 0 0
     HLOAD TEMP 8 TEMP 7 0
     JUMP L5
L4   ERROR
L5   NOOP
     MOVE TEMP 9 MINUS TEMP 1 1
     MOVE TEMP 10 CALL TEMP 8 (TEMP 0 TEMP 9)
     MOVE TEMP 11 TIMES TEMP 1 TEMP 10
     MOVE TEMP 4 TEMP 11
L3   NOOP
     RETURN TEMP 4
END
```

我们可以计算出来，活性分析时，基本块 B1 的 OUT 集合里是没有 TEMP 2 的，回推到 MOVE TEMP 20 0 这一句，TEMP 2 不在这个语句的 OUT 集合中，却在 DEF 集合中，所以这是一个死代码，即完全不会被使用的定值。这种死代码可以直接删除。

但是有一种死代码不能直接删除：

如 MOVE TEMP 10 CALL TEMP 8 (TEMP 0 TEMP 9)，如果它是死代码，也不能直接删除，因为它涉及到了过程调用，而过程调用可能会改变过程所在的对象的值。

我采取的办法是把此句分解成：

```
MOVE a0 TEMP 0
MOVE a1 TEMP 9
CALL TEMP 8
MOVE TEMP 10 v0
```

增加了四种产生式，这样即使消除死代码也只删除最后一句 MOVE TEMP 10 v0。

但是消除死代码这件事本身可能会产生新的死代码，如下例：

```
MOVE TEMP 20 0
MOVE TEMP 21 TEMP 20
```

假如之后的语句中既没有使用到 TEMP 20 的值也没有用到 TEMP 21。这个时候第一句不是死代码，第二句是，但是删除了第二句后第一句变成死代码了。

对于这种情况，一个比较笨的办法是，每次消除死代码后，重新对整个过程进行数据流分析，查找死代码。如此循环下去，直到不可再被优化。显然这种方法是不可取的。

我采取了与活性分析类似的方式，首先把所有基本块加入 Changed 集合，代表所有的基本块都要检查死代码；

循环检查 Changed 集合是否为空，若非空，取出一个基本块，重新生成 OUT 集合，倒推每行语句 IN/OUT 集合，消除死代码，然后检查其 IN 是否变动，如果变动，所有前驱结点加入 Changed 集合。

直到 Changed 集合为空，即到达不动点，所有死代码均消除。

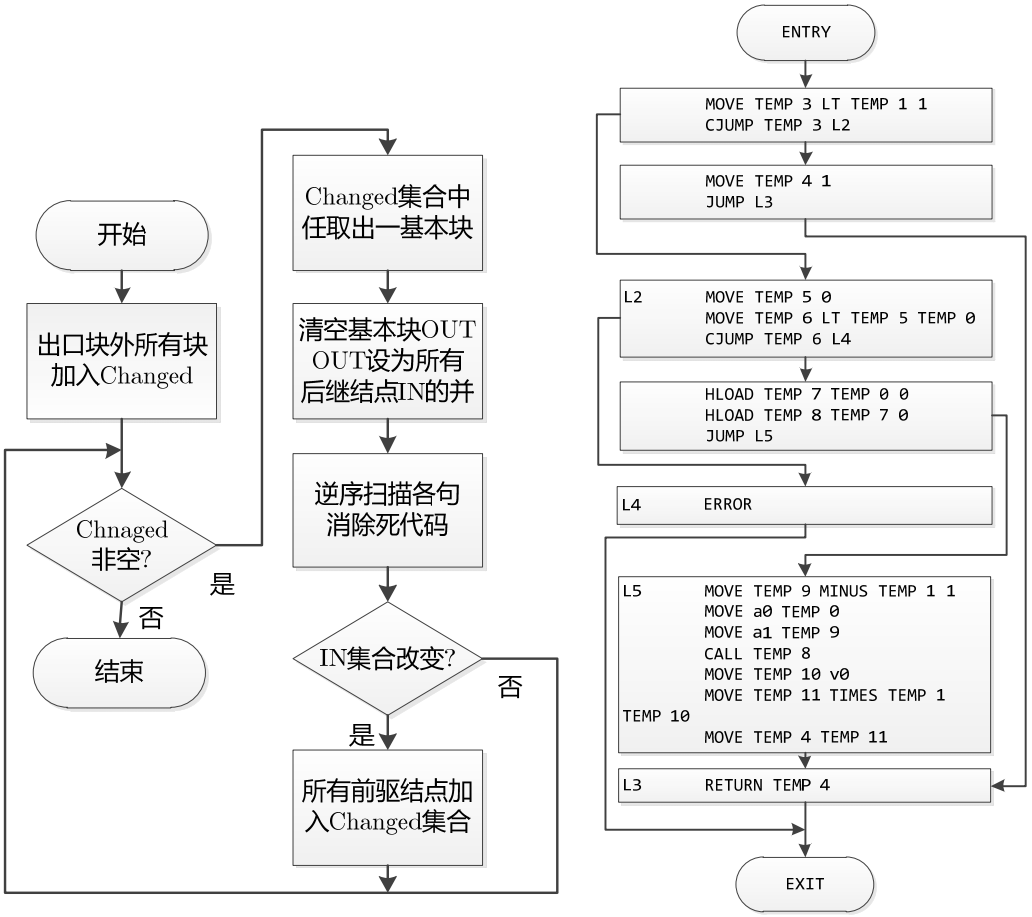
实现代码：

```
    //SMethod类removeDeadCode方法
    private void removeDeadCode() {
        HashSet<SBlock> lstChanged = new HashSet<SBlock>();//待检查的基本块集合
        lstChanged.addAll(lstBlock);//所有基本块都待检查
        while (!lstChanged.isEmpty()) {//未出现不动点
            SBlock block = lstChanged.iterator().next();//取一个基本块
            lstChanged.remove(block);
```

```
            if (block.removeDeadCode()) {//如果删除了新的死代码并且改变了Live-in
                for (SBlock prevBlock : block.getPrev()) lstChanged.add(prevBlock);
            }
        }
    }
    //SBlock类removeDeadCode方法
    protected boolean removeDeadCode() {//删除死代码，返回是否改变了Live-in
        lstLiveOut.clear();
        for (SBlock nextBlock : lstNext) lstLiveOut.addAll(nextBlock.lstLiveIn);
        HashSet<Integer> lstNextIn = lstLiveOut;
        for (int i = lstStmt.size() - 1; i >= 0; --i) {//逆序枚举每一语句
            SStatement stmt = lstStmt.get(i);
            HashSet<Integer> lstLiveOut = stmt.getLiveOut();
            lstLiveOut.clear();
            lstLiveOut.addAll(lstNextIn);//重置Live-out
            if (stmt.isDeadCode()) stmt.erase();//死代码变成NOOP,USE和DEF全部清空
            HashSet<Integer> lstLiveIn = stmt.getLiveIn();
            lstLiveIn.clear();
            lstLiveIn.addAll(lstLiveOut);//重新计算Live-in
            lstLiveIn.remove(stmt.getLiveDef());
            lstLiveIn.addAll(stmt.getLiveUse());
            lstNextIn = lstLiveIn;
        }
        HashSet<Integer> lstOldLiveIn = new HashSet<Integer>();
        lstOldLiveIn.addAll(lstLiveIn);//加入全部元素
        lstLiveIn.clear();
        lstLiveIn.addAll(lstNextIn);
        return lstLiveIn.size() != lstOldLiveIn.size()
|| !lstLiveIn.containsAll(lstOldLiveIn);//返回是否改变
    }
```

这里产生的 NOOP 语句将会被随后的清理中清理掉（除非某一个基本块所有语句都是 NOOP，这种情况只保留一个 NOOP）。

左下图是死代码消除算法的流程图，右下图为最终建立的 Fac_ComputeFac 的流图(SPiglet)：



5. Kanga → MIPS 汇编

5.1. 任务要求

将 Kanga 中间代码翻译成 MIPS 汇编代码，将 Kanga 语句翻译成对应的 MIPS 指令，并且实现对堆栈的操作。

重点：指令选择，维护堆栈

难度系数：3

5.2. 任务分析

本步骤作业难度不是很大，主要要做的是理解 MIPS 过程的堆栈的构成，实现对堆栈的操作；针对操作数的类型 (寄存器、标签、立即数) 选择不同的 MIPS 指令；通过调用系统调用来分配内存、打印信息、报错退出。

转换 MIPS 汇编的工作完全由 ConvertToMipsVisitor 类实现。

5.3. 堆栈维护

MIPS 的堆栈结构如下图所示：

高地址			低地址
ra	fp	Kanga 代码中的溢出栈	下一帧

假设 Kanga 代码中某过程的第二个参数为 X，即使用了溢出单元 X 个，翻译成 MIPS 汇编后，在进入该过程时，就应该将堆栈指针 sp 减去 4X+8 个以保存所有溢出单元及返回地址 ra 和帧指针 fp。(由于 MAIN 函数不需要保存帧指针，只需要减去 4X+4)。在过程结束时，sp 要加上 4X+8(MAIN 函数为 4X+4).

以 Fac_ComputeFac [2][5][2]为例，在进入过程时，输出：

```
sw $fp, -8($sp)          #保存帧指针
sw $ra, -4($sp)          #保存返回地址
move $fp, $sp            #原来的栈指针指向现在的帧指针
subu $sp, $sp, 28        #栈指针减去 28(=4*5+8)
```

离开进程时，输出：

```
addu $sp, $sp, 28        #栈指针加上 28(=4*5+8)，回复到原来的值
lw $fp, -8($sp)          #恢复帧指针
lw $ra, -4($sp)          #恢复返回地址
```

由于一个过程在被调用前，就可能被其他过程利用 PASSARG 传递参数，而其他过程是不知道被调用过程的溢出栈大小的，所以 Kanga 溢出单元应在 MIPS 溢出单元里倒序排列，这样被调用过程溢出单元 i 就可以通过-(12+4i)($sp)访问到。如 PASSARG 1 s0 就会被翻译成 sw $s0, -12($sp)(PASSARG 从 1 开始).

而一个过程访问自己的溢出栈时，由于溢出栈倒序排列，访问溢出栈 SPILLEDARG i 应该被翻译成 -(4(spillsize-i-1))($sp) ,即在 sp 指针指向的内存空间向高地址方向偏移 4*(总溢出栈个数-溢出栈编号-1)。例如 ALOAD s0 SPILLEDARG 0 语句，假设 Kanga 溢出栈长度为 5，则翻译为：lw $t4, 16($sp)。

5.4. 系统调用

需要用到系统调用的地方有：分配内存，打印，报错退出。均由 syscall 指令执行,寄存器 v0 决定功能，a0 传递参数，如果系统调用有返回值的话，由 v0 返回。

系统调用表：

功能	v0	参数	返回值
打印整数	1	a0=待打印整数	N/A
打印字符串	4	a0=待打印字符串首地址	N/A
分配内存	9	a0=分配字节数	v0=分配内存空间首地址
退出	10	N/A	N/A

ERROR 指令就被翻译成：

```
la $a0, error     #传入字符串，"ERROR: abnormal termination\n"
li $v0, 4         #功能号设为 4
syscall           #打印字符串
li $v0, 10        #功能号设为 10
syscall           #结束程序
```

5.5. 指令选择

这里主要要考虑的是，相同的 Kanga 指令，使用的表达式不同，可能会翻译成不同的 MIPS 代码。

以 MOVE Reg Exp 为例：

1) 假设 Exp 为一个 HAllocate，则 Visitor 访问 HAllocate 时，就已经将输出了系统调用语句，这时直接输出 move $Reg, $v0 即可；
2) 假设 Exp 为一个 SimpleExp：
   a) 若是一个 Reg，直接输出 move $Reg1, $Reg2
   b) 若是一个 IntegeralLiteral，即立即数，输出 li $Reg, 数字
   c) 若是一个 Label，即标签，输出 la $Reg, 标签
3) 假设 Exp 是一个 BinOp，即整个 Kanga 语句是 MOVE Reg1 Operator Reg2 SimpleExp，考虑操作符的种类，SimpleExp 的类型：
   a) 若 SimpleExp 是一个 Reg：

      Operator 为 LT：输出 slt $Reg1, $Reg2, $Reg3

      Operator 为 PLUS：输出 add $Reg1, $Reg2, $Reg3

      Operator 为 MINUS：输出 sub $Reg1, $Reg2, $Reg3

      Operator 为 TIMES：输出 mul $Reg1, $Reg2, $Reg3

   b) 若 SimpleExp 是一个 Label：

      先把这个 Label 移到一个临时寄存器中，一般选择 v0，但是如果 Reg2 也是 v0(即读取溢出的 SPiglet 临时单元)，那么就把 Label 移到 v1.

      设这个临时寄存器为 Reg3，先输出：la $Reg3, 标签

      然后就可以把 SimpleExp 看做 Reg3，同 MOVE Reg1 Operator Reg2 Reg3 情况处理；

   c) 若 SimpleExp 是一个 IntegerLiteral：

      将数字利用 li 伪指令移动到一个临时寄存器去，选取临时寄存器的方法与 b)中相同，之后同 MOVE Reg1 Operator Reg2 Reg3 处理。

      注意不要使用立即数指令 slti/addi，因为这些指令仅支持 16 位(-32768-32767)整数，若超出范围即报错，而利用 li 伪指令可以避免这个问题。

三、工具软件与测试

1. 工具软件

本作业在词法/语法分析时 ,使用了 JavaCC/JTB 对给定的 BNF 范式(巴科斯-诺尔范式)自动生成词法、语法分析器。在编写代码时主要使用了 Eclipse 集成开发环境，测试时 Piglet,SPiglet 代码由 Piglet Interpreter 解释执行，SPiglet 的语法正确性由 SPiglet Parser 检查。Kanga 代码由 Kanga Interpreter 解释执行。最后产生的 MIPS 汇编代码可以运行在 PC SPIM 模拟器上。

1.1. JavaCC/JTB

JavaCC/JTB 是针对给定的 BNF 范式，产生对应的词法、语法分析器（Parser）的程序。其中 JTB 会将 BNF 范式（存在.jj 文件中）转换为抽象语法树（AST，保存在.out.jj 文件中），之后再利用 JavaCC 将抽象语法树转成 syntaxtree 和 visitor 的 Java 代码。

用法 :(以建立 MiniJava Parser 为例)

JTB: java -jar jtb.jar minijava.jj (minijava.jj 为 MiniJava 之 BNF 范式文件)

Java CC: javacc jtb.out.jj (jtb.out.jj 为上一句指令之生成文件)

之后产生了 syntaxtree 和 visitor 的 Java 代码，直接使用即可。

优点 :基本上完成了所有的词法/语法分析过程 ,用户只需要书写语法制导翻译即可。产生的 syntaxtree 和 visitor 直观性很强。

缺点：无。

1.2. Eclipse

编写编译实习作业的 IDE，使用方法略。

优点：代码提示功能强大。

缺点：稳定性非常差，经常无响应甚至直接崩溃，或者在明明没有任何代码错误的情况下提示错误拒绝编译。

比如，我在类型检查时，在 BuildSymbolTableVisitor 使用了 MMethod 类（MiniJava 方法类），结果 Eclipse 在 MMethod 这个类标识符的后六个字母 Method 的下画了条红色横线(MMethod)，提示找不到”Method”标识符。不得已我将源代码拷出，删除并重新创建工程再拷回代码，问题才得以解决。

此外，在调试的过程中，Eclipse 的变量监视(Watch)常常会自己崩溃掉，甚至是 Eclipse 本身（而非我的代码）抛出了一个异常。

遗憾的是，似乎在 Java 的集成开发环境中，并没有如 Visual Studio 和 RAD Studio 一样稳定、可靠、好用的软件，Microsoft 和 Borland 曾经先后推出过 Visual J++和 Borland JBuilder，但是由于和 Sun 的版权问题，这些项目最后都被放弃了。

1.3. Piglet Interpreter

Piglet 代码的解释器。

用法 :(以 Factorial.pg 为例)将 Piglet Interpreter 的压缩包解压(老师提供的是 rar ,所以必须解包)，然后在 PgInterpreter.class 文件所在的目录执行：

java PgInterpreter < Factorial.pg

解释器将会输出结果。

优点：用法相对简单，直接传入 Piglet 代码文件即可。可以支持命令行操作，便于批处理自动测试。

缺点 : 在编译实习作业中没有什么缺点，但是我在本学期的另一门课 Web 技术概论中，将编译实习的作业做成了可以在线执行的 Servlet，由于 Piglet Interpreter 在解析 ERROR 命令时会直接调用 System.exit(0);指令，导致 Tomcat 软件崩溃，且使用了过多的静态变量，导致多次执行该程序后直接

ArrayOutOfBoundException，我不得不修改解释器的代码。

1.4. SPiglet Parser

检查生成的 SPiglet 代码是否严格符合 SPiglet 的产生式（由于 SPiglet 语言是 Piglet 语言的子集，代码可以直接在 Piglet Interpreter 上解释执行）

用法：（以 Factorial.spg 为例）将 SPiglet Parser 的压缩包解压，然后在 Main.class 文件所在目录执行：

java Main < Factorial.spg

如果符合 SPiglet 语法，将输出：Program parsed successfully

优点：可以支持命令行操作，便于批处理自动测试。

缺点：无。

1.5. Kanga Interpreter

Kanga 代码的解释器。

用法：（以 Factorial.kg 为例）将 Kanga Interpreter 的压缩包解压，然后在 kgi.class 所在目录执行：

java kgi < Factorial.kg

解释器将会输出结果。

优点：可以支持命令行操作，便于批处理自动测试。

缺点：基本无法使用。

由于 kgi 将所有的临时单元人为地分为堆类型和数值类型（实际上应该都是整数），对于表达式 MOVE TEMP 0 LT TEMP 1 TEMP 2，若 TEMP 1 为堆类型（如对象、数组），整个 LT 表达式的类型都将是堆类型，导致 TEMP 0 变为堆类型。而 kgi 解释 CJUMP TEMP 0 LABEL 时，若发现 TEMP 0 是堆类型，会直接报错退出。

由于我之前检查数组和对象的空指针引用，会先比较对象（即所谓堆类型临时单元）地址和 0 的，然后根据比较结果（堆类型）决定跳转。所以我自己的编译器产生的 SPiglet 代码在转换为 Kanga 后无法在 kgi 上执行。（但是老师提供的没有进行数据安全性检查的 SPiglet 代码测试样例可以正常执行）。

此外，kgi 禁止一次性分配超过 40 个字节的内存空间。

1.6. PC SPIM

MIPS 模拟器，可以解释执行 MIPS 汇编，并观察各寄存器。

用法：将产生的 MIPS 汇编代码保存到一个.s 或.asm 文件中，用 SPIM 打开，按 F5 执行，控制台将会输出 MIPS 汇编的运行结果。

优点：对 MIPS 汇编支持较为全面，而且提供寄存器数据观察，便于差错。

缺点：

控制台输出的程序执行结果无法复制，必须保存到 PCSpim.log 日志，日志附加过多信息，不利于结果的比较。

窗口界面，非控制台界面，难以用批处理来批量检查 MIPS 汇编结果的正确性。

界面友好性差。

2. 测试用例和方法

2.1. 构建测试用例

一般来源于 UCLA 提供的八个程序的代码，或者经过自己的编译器转换输出的中间代码，或 Hundred Hour Wood Showcase(http://hundredhour.appspot.com)提供的代码（此网站部分代码转换不正确）。此外也有一些自己书写的代码，一般较短，对几个可能有问题的地方进行针对性的检查。

2.2. 测试方法

手动测试与批量测试均有，下为一批量处理文件，用来测试 Piglet 转换 SPiglet 结果是否正确：

```
@echo off
E:
cd "E:\Courses\3A\Practice for Compiler Design"
type piglets\%1.pg > a.tmp
cd "E:\Programming\Workspace\PCD\bin"
java piglet.piglet2spiglet.Main < "E:\Courses\3A\Practice for Compiler Design\a.tmp" >
"E:\Courses\3A\Practice for Compiler Design\b.tmp"
cd "E:\Courses\3A\Practice for Compiler Design\spp"
java Main < ..\b.tmp
cd ..\pgi
java PgInterpreter < ..\a.tmp > a.tmp
java PgInterpreter < ..\b.tmp > b.tmp
fc a.tmp b.tmp
cd ..
```

假设有 Factorial.pg 存于 E:\Courses\3A\Practice for Compiler Design\piglets 目录，在此目录的上级目录执行 test Factorial，输出结果：

Program parsed successfully

Comparing files a.tmp and B.TMP

FC: no differences encountered

即输出 SPiglet 代码为合法的 SPiglet 代码，且执行结果与原 Piglet 代码相同。

3. 测试发现的错误

测试效果较好，特别是在最后的书写报告时，经过对代码重新分析、思考，发现了如下两个错误：

3.1. 循环继承检查

在类型检查模块，检查循环继承时，我一开始的代码是：

```
    String szBases = newClass.getName();
    while (szBase != null) {
        szBases += "->" + szBase;
        if (szBase.equals(newClass.getName())) {//出现循环继承
            szBases = szBases.substring(0, szBases.length());
            classList.addError(new TypeError(n.f3.f0.beginLine, "Circular extends: " + "\""
+ szBases  + "\""));
            break;
        }

        MClass baseClass = classList.get(szBase);
        if (baseClass != null) szBase = baseClass.getBase();
        else break;
    }
```

利用表达式 szBase.equals(newClass.getName())来判断是否发生循环继承。它只判断某个类的基类中是否出现了自己，即 A→B→A。但是后来我构造了一个样例，有 A,B,C 三类，其中 A→B→A，C→A（A,B 互相循环继承，C 类为 A 类子类）

```
class Test {
    public static void main(String[] a){
        System.out.println(new A().start());
    }
}

class A extends B {
    public int start(){
        return 0;
    }
}
class B extends A { }
class C extends A { }
```

原来的程序在判断 A,B 类时能正确处理，输出循环继承错误，但是判断 C 类时，会陷入死循环中。所以在检查一个类的基类时，应该将这个类所有的基类保存到一个集合中，当新的基类已经处在这个集合中时，跳出这个循环，更改后的代码已在前面类型检查一章 1.6 节 2)循环继承检查部分贴出，上述测试样例

存在的类型错误将输出为：

```
Line 7: Circular extends: "A->B->A"
Line 12: Circular extends: "B->A->B"
2 type error found.
```

但是只有这个新的基类是原来的类本身时才报循环继承错误，如 A→B→A 和 B→A→B，而在基类内部出现循环时，如 C→A→B→A，C 类定义处不需要报循环继承错误。

3.2. 死代码消除

一开始我采用了《现代编译器的 Java 实现》第 17 章中增量式数据流分析的办法，针对每一条被删除的死代码，形如 MOVE TEMP 0 TEMP 1，由于 TEMP 0 在该语句活性分析的 DEF 集合中，TEMP 1 在 USE 集合中。该句被删除后，在这一语句中存在对 TEMP 0 的定值和对 TEMP 1 值的使用将被取消。

因为死代码的定义是：如果在语句的 DEF 集合中的元素不在其 OUT 集合中，即该语句定值的临时单元的值从未被使用过，所以该定值语句为死代码。所以取消掉对 TEMP 0 的定值后，由于 OUT 集合中本来就没有 TEMP 0，所以 IN 集合也不会有 TEMP 0，所以定值直接取消即可，对前面的语句没有影响。

如果被删除的死代码语句的 OUT 集合中存在 TEMP 1，那么即使死代码语句中对 TEMP 1 值的使用被取消，IN 集合中仍然会有 TEMP 1，所以对前面的语句仍然不会有影响。

如果被删除的死代码语句的 OUT 集合中无 TEMP 1，如果语句中对 TEMP 1 的使用被取消，那么 IN 集合中的 TEMP 1 将会被删除，并且回推到以前的语句。对于之前的每一个语句，从 IN 和 OUT 集合中删除 TEMP 1，直到前一个使用了 TEMP 1 的值的语句为止。

可能是我对增量式数据流分析的理解有所偏差（也可能是此书中文翻译版本确实太过难以让人理解），我写的算法针对一个基本块内的语句是可以正常使用的，但是如果超出了一个块的范围，就会出现问题。

构造测试样例：

```
     MOVE TEMP 1 0
     CJUMP TEMP 2 L0
     MOVE TEMP 0 TEMP 1
     JUMP L1
L0:  PRINT TEMP 1
L1:  后面的语句，其中没有对TEMP 0和TEMP 1值的使用
```

可以分块为 3 块（不考虑 L1 及之后的语句）：

```
     MOVE TEMP 1 0
     CJUMP TEMP 2 L0
```

```
     MOVE TEMP 0 TEMP 1
     JUMP L1
```

```
L0:  PRINT TEMP 1
```

这三个块依次编号为 B1，B2，B3。其中，B1 是 B2 和 B3 的前驱结点，那么如果 B2 中删除了死代码 MOVE TEMP 0 TEMP 1，由于这条语句的 OUT 集合中无 TEMP 1，所以回推以前语句，在 IN 和 OUT 集合中删除 TEMP 1。当回推删除 TEMP 1 到 B1 块内时，我一开始设计的算法会接着直接删除 B1 各语句中 IN 和 OUT 集合的 TEMP 1（直到前一次对 TEMP 1 的值的使用）。

当回推到第一句 MOVE TEMP 1 0 时，由于此语句 OUT 集合中 TEMP 1 被删除，DEF 集合有 TEMP 1，被认为是死代码而将被删除，这将造成 B3 中 PRINT TEMP 1 的 TEMP 1 无定义。

为了解决这个问题，必须重新统计整个 B1 基本块的 OUT 集合。经过思考，我重写了死代码消除算法，采用了类似于活性分析的迭代—寻找不动点算法消除死代码。（但是绝不是消除一次死代码就重新活性分析的笨办法）此算法在前面 SPiglet 转换 Kanga 一章 4.12 小节中已详细介绍。

以上两个错误分别发现在类型检查和 SPiglet→Kanga 的机器测试后，但这两次机器测试似乎均未扣分。（最后总分均在最高分数区间 14~15）可能是测试样例的强度不够，没有针对这些错误的检查。

四、总结

1. 作业总结

本次作业较好的完成了类型检查, MiniJava→Piglet, Piglet→SPiglet, SPiglet→Kanga, Kanga→MIPS 汇编五步，在五次的面测和机器测试总分均在最高的分数区间(14~15)内，程序可靠性较好。

为了保证程序访问数据的安全性，在代码运行时（非类型检查时），对所有对象/数组的访问都进行了空指针检查，数组下标进行了上下越界检查，创建数组长度也进行了下越界检查(必须为正数)，如果出错，程序报错后退出。这样就能阻止目标代码访问不该访问的内存空间。

对整个程序进行了建立了流图并进行了到达定义、活性分析两种数据流分析，通过将中间代码转换为静态单赋值形式，优化了寄存器分配的线性扫描算法，而且消除了死代码和一些不可能被访问到的代码，减少了寄存器和溢出单元的使用，缩短了生成的目标代码。

2. 心得体会

上学期我学习了编译原理课，虽然学得还可以，但是对于编译技术的很多知识都只是一个感性的认识，并没有在实际操作中的运用到它们，这学期的编译实习课给了我这样一次机会。

老实说，编译实习课很花时间，而且在某些模块，如类型检查和 SPiglet 转换 Kanga 有一定的难度。因此，很多同学选择了中途放弃。但是我觉得，相比于所花费的时间和精力，编译实习课能给我们带来更大的收获。上学期很多理解不够深入、似懂非懂的地方能够通过编译实习的训练得到了更为深刻的理解。更重要的是，随着编译器功能的一步步完善，原本的高级语言 MiniJava 代码被改写成越来越接近底层的中间代码，当其最终被转换成 MIPS 汇编时，那种成功的愉悦感（特别是不组队一个人独立完成时）是难以形容的，而这种情感又能反过来激发我们更多兴趣，来对编译器进行进一步地修改、优化。

编译实习课是一门综合锻炼多种技能的课，如类型检查、数据流分析、寄存器分配算法是考验我们的算法能力，而编译器的实现又考验了我们的代码实现能力。编译器写出来后，如何保证它能完美的通过所有的测试点，这个又考验到了我们的代码调试能力。甚至，由于编译实习的时间花费很大，它还考验了我们时间安排和统筹规划能力——如何合理安排写编译和学习其他科目的时间。

有人说计算机系大三上学期是一个很“虐”的学期，诚然，这学期专业课很多，而且大多很花费时间，但是这不代表我们就有权利消极怠慢，甚至半途而废。也许这种“虐”本身就是一种历练，无论是知识、能力上，还是意志上。

3. 课程建议

1. 编译原理课在介绍语法制导定义、中间代码生成、目标代码生成和代码优化前，还介绍了词法分析和语法分析，而在编译实习中，这两部分内容完全由 JavaCC/JTB 帮我们实现了，建议老师可以适当增加一些有关的实习内容（比如以代码填空的形式考察 MiniJavaParser 源代码）；

2. 时间分配略不合理，Piglet→SPiglet 和 Kanga→MIPS 很简单，而 SPiglet→Kanga 却很难，但是这三步都各分配了两周时间，结果前两种转换一天之内就能写完，后者时间却不是很够（特别是这学期还和两门期中考试冲突）。建议把 Piglet→SPiglet 和 Kanga→MIPS 改为要求一周完成，而 SPiglet→Kanga 给的时间延长到 3 周。

4. 注意事项

1. 本次作业在 JDK 1.7 Update 9 版本下编译调试通过。如果移到更低的 JRE/JDK 版本，由于 Java 语言特性的改变，可能需要更改源代码。

2. 本作业有在线版本 MiniJava Online Interpreter:http://162.105.203.19:8078/hw/5/interpreter.html，由于这是 Web 技术概论课的作业网站，这学期结课后可能会被关闭。但是截止到我上交该报告为止此网页仍可以正常访问。使用方法为：在文本框中输入 MiniJava 代码，点击 Check!按钮，等待一段时间后（略长，但不超过 1 分钟），网页将显示 MIPS 汇编代码及运行结果。