# 编译实习报告

鲁云龙　1600011045

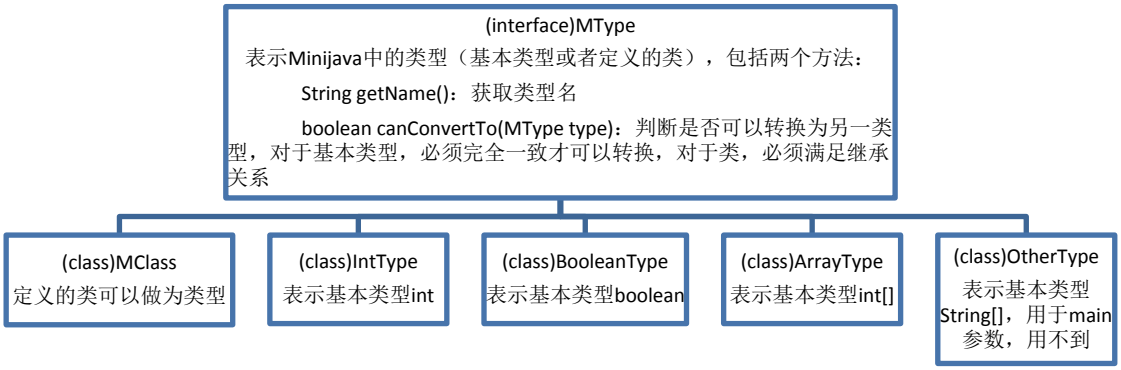
2018/6/20

# 一、Minijava 类型检查

整个 Minijava 编译器的第一步是对输入的 Minijava 源程序进行类型检查。类型检查的任务主要是：对每个类、类中的字段与方法、方法的参数与局部变量建立符号表，并检查所有对类、方法和变量的使用是否符合类型规范。由于这一步中主要是对类型的检查与分析，因此本次作业主要进行了以下相关建模和翻译的步骤：

1.**Minijava 的符号表模型**为：


此外为了能表示 Minijava 中的“类型”，建立一个接口 MType，其继承关系如下：




在这种建模下，后期对类型的检查完全对于变量 MVar 的类型 MType 进行，不需要操作底层的字符串，耦合性很低。

2.**类型错误的处理机制**

建立 TypeCheckError 类封装一个类型错误，继承自 Error，其中包含错误的信息和出现的位置（所在行列）。使用一个 ErrorManager 进行统一管理，出现错误时调用 ErrorManager.error（TypeCheckError error）方法将错误交给 ErrorManager 处理，对于我们的程序直接将错误信息打印并退出即可。

3.**建立符号表**

符号表使用一个类 SymbolTable 加以管理，该类中维护一个类的 HashMap 储存所有声明的类，以及一个表示 main 方法的 MMethod，并且都使用 static 作为类静态成员。有一点要注意的是，为了快速的检索，无论是 SymbolTable 中管理类，还是 MClass 中管理方法 MMethod、字段 MVar 列表，以及 MMethod 中管理局部变量 MVar 列表，均使用 HashMap 储存名称到对象的映射，这样使用名称可以快速找到对应的对象。

建立符号表时会遇到一个问题，就是在一次遍历过程中，如果当前类继承的父类或者当前变量的类型还没有被声明，那么就没有办法将其设置为对应的 MClass 对象，而是只能通

过一个类名 String 标记其类型，这会使得建立的符号表较为混乱，难以使用上述的模型类进行统一管理。因此，我的程序使用两个 Visitor 建立符号表。第一个 Visitor 只遍历所有类的声明部分，将得到的 MClass 对象加入到符号表中。第二个 Visitor 遍历每个类内部，记录其父类信息、字段和方法信息以及方法的参数和局部变量信息。这两个 Visitor 相结合就可以完成整个符号表的建立。

（1）ClassVisitor：负责将每个类的声明转化为 MClass 对象加入符号表。由于本次遍历不需要向下传递作用域信息，因此使用 GJNoArguDepthFirst 基类 Visitor。需要返回值是因为在遍历 Identifier 词法节点时需要将标识符包装成 MIdentifier 返回到上层（比如用做类名）。

在这一次遍历中可以检查出的类型错误有：类的重复定义。

（2）MemberVisitor：负责进行符号表剩余部分的建立。本次遍历中会涉及到类中方法的遍历和方法中变量的遍历，需要将作用域信息向下传递。同时在访问 Identifier 词法节点时，使用其字符串在当前作用域（可能是类的字段或方法的局部变量）中找到 MVar 对象，如果找不到则报类型错误，找到就返回对应的 MVar 对象。在访问 Type 词法节点时，需要根据类型返回相应的 MType 对象（要么是 MClass 要么是基本类型）。因此该 Visitor 继承自 GJDepthFirst 基类 Visitor。

在这一次遍历中可以检查出的类型错误有：类型未定义、变量重复定义、方法重复定义。

**4.处理字段和方法继承**

在使用两个 Visitor 建立了符号表之后，此时并没有处理类继承时字段和方法的继承，而是只记录了每个类的父类。接下来的一步就是在已经建立的符号表内部进行类字段和方法的继承，主要分为两步：先对于类的继承关系图进行拓扑排序，然后根据拓扑顺序处理每一组类的继承，将父类的变量和方法拷贝到子类符号表中。

为了进行类继承关系图的拓扑排序，程序中建立了一个工具类 Graph 表示一个图，可以往其中添加节点、添加边以及进行拓扑排序等操作。若拓扑排序成功则返回一个合法的拓扑顺序，否则报错，即此时继承关系中出现了环。

在得到了类继承的拓扑顺序后，按顺序处理每一个类，对其父类的字段和方法进行拷贝。这里需要注意的是，如果子类一个方法名与父类相同但签名不同，则报错（方法重载错误），否则直接复制，但要注意子类的字段和方法优先，即可以覆盖父类中的同名字段或方法。

在这一步中可以检查出的类型错误有：类的循环继承、错误的方法重载。

**5.进行语句的类型检查**

在前面的步骤中我们已经把整个符号表建立完毕，此时只需要一个 Visitor 遍历所有的 Minijava 语句，分析其中的变量与表达式在语句中的类型是否匹配即可。由于符号表中所有的类型都是用 MType 表示的，在判断类型匹配时直接调用相应 MType 的 canConvertTo 方法即可，代码十分简洁。

该 Visitor 遍历时需要将作用域信息（MClass 或者 MMethod）往下传。同时对于 Identifier 词法节点，需要从当前作用域的符号表中查找指定的方法或变量，如果找不到则报错（变量或方法未定义），否则返回相应的 MMethod 或 MVar 对象。以及对于表达式的词法节点 Exp，对于每种具体的表达式需要将其返回值类型包装成一个 MVar 往回传。因此本次 Visitor 继承自 GJDepthFirst 基类 Visitor。

在这一次遍历中可以检查出的类型错误有：使用未定义的类型、方法、各种语句和表达式中的类型不匹配、方法调用的参数不匹配以及方法的返回类型与签名不匹配。

附：一些关键代码的实现

拓扑排序相关代码：

```java
/**
 * 对图进行拓扑排序，如果排序失败（存在环）则抛出类继承的异常，到上层进行处理。
 * @return 若排序成功，返回各节点的一个合法的拓扑顺序
 * @throws TypeCheckError 类继承的异常
 */
public ArrayList<T> topSort() throws TypeCheckError {
    ArrayList<T> ans = new ArrayList<>();
    Queue<T> q = new LinkedList<T>();
    for (Entry<T, Integer> entry : degrees.entrySet())
        if (entry.getValue() == 0)
            q.add(entry.getKey());
    while (!q.isEmpty()) {
        T head = q.remove();
        ans.add(head);
        HashSet<T> next = graph.get(head);
        for (T node : next) {
            int newDegree = degrees.get(node) - 1;
            degrees.replace(node, newDegree);
            if (newDegree == 0)
                q.add(node);
        }
        graph.remove(head);
    }
    if (!graph.isEmpty()) {
        MIdentifier cycleElement = graph.keySet().iterator().next();
        throw new TypeCheckError(String.format("Cycle detected: a cycle exists in the type hierarchy of '%s'",
                cycleElement.getName()), cycleElement.getRow(), cycleElement.getColumn());
    }
    return ans;
}
```

处理类字段和方法继承的相关代码：

```java
/**
 * 继承父类的域和方法
 */
public void copyFromSuper() {
    if (superClass == null)
        return;
    for (Entry<String, MMethod> entry : methods.entrySet()) {
        String name = entry.getKey();
        MMethod method = entry.getValue();
        /*
         * Check: Is there a method in super class with the same name but different
         * signature?
         */
        MMethod m = superClass.findMethod(name);
        if (m != null && !m.equals(method))
            ErrorManager.error(
                    new TypeCheckError(String.format("Method overriding fails: incompatible signature with '%s.%s'",
                            m.getScope().getName(), name), method.getRow(), method.getColumn()));
    }
    HashMap<String, MVar> newFields = (HashMap<String, MVar>) superClass.fields.clone();
    newFields.putAll(fields);
    fields = newFields;
    HashMap<String, MMethod> newMethods = (HashMap<String, MMethod>) superClass.methods.clone();
    newMethods.putAll(methods);
    methods = newMethods;
}
```

类型检查的简洁包装：

```java
/**
 * 检查变量{@code exp}是否为类型{@code type}
 * @param exp 右值表达式
 * @param type 指定类型
 */
private void typeCheck(MVar exp, MType type) {
    if (!exp.getType().canConvertTo(type))
        ErrorManager.error(new TypeCheckError(String.format("Type mismatch: cannot convert from '%s' to '%s'",
                exp.getType().getName(), type.getName()), exp.getRow(), exp.getColumn()));
}
```

类型检查举例：数组赋值语句中，检查数组为 ArrayType，下标和右值表达式为 IntType。

```java
/**
 * <p>对于数组赋值语句，需要检查数组类型、下标为{@code int}类型，以及右侧表达式为{@code int}类型
 * <p>f0 -> Identifier()
 * f1 -> "["
 * f2 -> Expression()
 * f3 -> "]"
 * f4 -> "="
 * f5 -> Expression()
 * f6 -> ";"
 */
public Object visit(ArrayAssignmentStatement n, MIdentifier argu) {
    MVar var = (MVar) n.f0.accept(this, argu);
    typeCheck(var, MType.ArrayType);
    MVar index = (MVar) n.f2.accept(this, argu);
    typeCheck(index, MType.IntType);
    MVar exp = (MVar) n.f5.accept(this, argu);
    typeCheck(exp, MType.IntType);
    return null;
}
```

# 二、从 Minijava 到 Piglet

在对 Minijava 进行类型检查后，编译器下一步是将 Minijava 源程序翻译成第一种中间语言 Piglet。**中间语言 Piglet 有以下特点：**

（1）面向过程而非面向对象，代码只由一个个的过程组成，程序入口为 main 过程。原先类方法的传参需要加上类对象本身作为第一个参数。

（2）有无限多的临时储存单元。

（3）所有表达式均为前缀形式，允许表达式的嵌套。

（4）调用函数传参时最多只能使用 20 个临时储存单元传递参数。

**本次翻译过程主要思路及注意点如下：**

（1）关于 Minijava 中变量的处理：

int,boolean 都看做 Piglet 中的整数，其中 true 看做 1，false 看做 0；

int[]在 Piglet 中使用一个地址来表示，该地址指向一块被分配的空间，开头四字节是数组长度 n，后面 4n 字节是数组的实际内容。

Minijava 中的类实例在 Piglet 中使用一个地址来表示，该地址指向一块被分配的空间，开头四字节是一个地址，指向该实例的类的方法表，后面内容是该实例的属性表。其中方法表中储存原类的各个方法的地址，属性表中储存该实例的各个字段的值。

（2）关于方法的多态和属性的覆盖：

由于子类完全继承父类的方法，因此让子类的方法表前面一半与父类方法表顺序保持一致，若有重载的方法，替换相应的方法地址即可；子类新增的方法放在方法表中后一半。

由于子类同名属性仅仅是覆盖父类属性，在调用父类方法时仍然可能访问到父类被覆盖的属性，因此对于子类和父类的重名属性不能进行替换，而是对子类所有的属性都开辟额外的空间。也就是说，子类属性表中前一半是父类的属性表，后一半是子类的所有字段（重名和新增的）。

（3）关于变量的初始化：

类的成员变量在实例化该类，分配属性表时，全部初始化为 0；

方法的局部变量在进入该方法时全部初始化为 0；

数组的内容在分配该数组时全部初始化为 0。

（4）关于具体语句翻译的注意点：

在使用引用型变量（类或数组）时，需要先检查非空；

在对数组进行下标操作时，需要检查下标是否越界；

在新建数组时，数组长度必须非负；

对于&&短路与运算符，当左操作数为 0 时，不计算右操作数。

以下是具体翻译过程：

**1.建立符号表**

建立符号表的过程与 Minijava 类型检查时建立符号表的过程相同，都是先通过一个 ClassVisitor 将所有类加入符号表，然后使用 MemberVisitor 将类成员、方法及局部变量等加入符号表，最后在符号表内部进行类继承的拓扑排序并按顺序进行字段和方法的拷贝。

与类型检查不一样在于，由于在操作类实例的属性表和方法表时，需要找到每个方法和属性在表中的位置，因此在类 MMethod 和 MVar 中各添加了一个字段 id。对于 MMethod 而言，id 表示该方法在类的方法表中的索引下标；对于 MVar 而言，若该 MVar 为参数，则 id 表示该参数是第几个参数（从 0 开始，方便访问对应 TEMP），若该 MVar 为类的字段，则 id 表示该字段是第几个字段（从 1 开始，方便计算对应属性表的位置）。

此外在进行类字段和方法的继承时，对新增的字段和方法重新分配 id 从而实现方法的多态和属性的覆盖。

**2.关于 Piglet 输出格式的规范**

由于 Piglet 代码可能嵌套表达式，因此如果缩进不好的话，Piglet 代码将毫无可读性。下图给出了不良好缩进的 Piglet 代码（ucla 样例 TreeVisitor.pg）和良好缩进的 Piglet 代码（我的程序对于 TreeVisitor.java 的翻译结果）

```
    HSTORE TEMP 59  12 Tree_GetRight
    HSTORE TEMP 59  8 Tree_SetLeft
    HSTORE TEMP 59  4 Tree_SetRight
    HSTORE TEMP 59  0 Tree_Init
    MOVE TEMP 61  4
L2  CJUMP  LT TEMP 61  28 L3
    HSTORE  PLUS TEMP 60 TEMP 61  0  0
    MOVE TEMP 61  PLUS TEMP 61  4
    JUMP L2
L3  HSTORE TEMP 60  0 TEMP 59
 RETURN
TEMP 60
END

    MOVE TEMP 21  CALL
 BEGIN
    MOVE TEMP 64 TEMP 20
    HLOAD TEMP 62 TEMP 64  0
    HLOAD TEMP 63 TEMP 62  0
 RETURN
TEMP 63
END
(TEMP 64  16 )
    MOVE TEMP 21  CALL
 BEGIN
    MOVE TEMP 67 TEMP 20
    HLOAD TEMP 65 TEMP 67  0
    HLOAD TEMP 66 TEMP 65  72
 RETURN
TEMP 66
END
(TEMP 67 )
    PRINT  100000000
    MOVE TEMP 21  CALL
 BEGIN
```

```
MAIN
      PRINT CALL BEGIN
                 MOVE TEMP 55 BEGIN
                              MOVE TEMP 56 HALLOCATE 4
                              HSTORE TEMP 56 0 TV_Start
                              MOVE TEMP 57 HALLOCATE 4
                              HSTORE TEMP 57 0 TEMP 56
                            RETURN TEMP 57
                            END
                 CJUMP PLUS TEMP 55 1 L0
                 ERROR
            L0   HLOAD TEMP 53 TEMP 55 0
                 HLOAD TEMP 54 TEMP 53 0
            RETURN TEMP 54
            END
            (TEMP 55)
END
```

```
                 L1   CJUMP LT TEMP 60 28 L2
                      HSTORE PLUS TEMP 59 TEMP 60 0 0
                      MOVE TEMP 60 PLUS TEMP 60 4
                      JUMP L1
                 L2   HSTORE TEMP 59 0 TEMP 58
                 RETURN TEMP 59
                 END
MOVE TEMP 21 CALL BEGIN
                      MOVE TEMP 63 TEMP 20
                      CJUMP PLUS TEMP 63 1 L3
                      ERROR
                 L3   HLOAD TEMP 61 TEMP 63 0
                      HLOAD TEMP 62 TEMP 61 0
                 RETURN TEMP 62
                 END
                 (TEMP 63 16)
MOVE TEMP 21 CALL BEGIN
                      MOVE TEMP 66 TEMP 20
                      CJUMP PLUS TEMP 66 1 L4
                      ERROR
                 L4   HLOAD TEMP 64 TEMP 66 0
                      HLOAD TEMP 65 TEMP 64 72
                 RETURN TEMP 65
                 END
                 (TEMP 66)
PRINT 100000000
MOVE TEMP 21 CALL BEGIN
                      MOVE TEMP 69 TEMP 20
                      CJUMP PLUS TEMP 69 1 L5
                      ERROR
                 L5   HLOAD TEMP 67 TEMP 69 0
                      HLOAD TEMP 68 TEMP 67 48
                 RETURN TEMP 68
                 END
                 (TEMP 69 8)
```

的对比：

为此我的程序定义了以下 Piglet 缩进规范：

（1）对于 Goal 和 StmtExp 中的 StmtList，须缩进一个 Tab 宽度，若 StmtList 中某语句前有 Label，则 Label 左端与缩进前对齐。

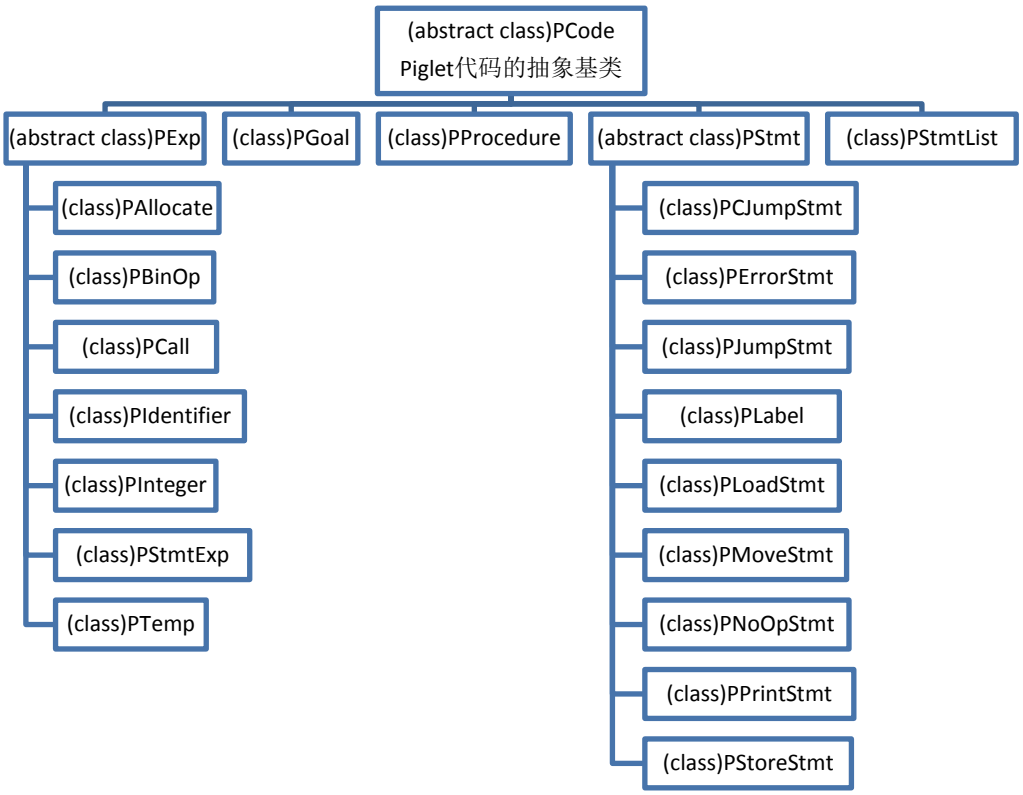
（2）一个 Piglet 表达式若跨越多行，每一行必须与上一行开头对齐，即整个表达式在代码中应处于一个矩形区域内（如右图）。

（3）一个 Piglet 语句有多个参数时，若全为单行参数，则写在一行内，否则对于每一个多行的参数，其后面一个参数应与第一个参数开头对齐，这样保证了语句的参数列表处于一个矩形区域内（如右图）。

**3.Piglet 代码块的包装与输出方式**

为了实现这样的缩进方式，本程序并不是现翻译现输出代码，而是翻译时构造了一个 Piglet 的语法树，再遍历该语法树从而自定义输出时的缩进方案。为此创建了一系列 Piglet

代码的包装类，其继承结构如下图所示：




其中抽象基类 PCode 定义了子类需要遵循的缩进规范与一些工具函数(如 intent 用于缩进，space 用于输出空格等)，每个子类重写了 int print(int depth)方法从而实现了对缩进的精确控制。要注意的一点是并非每个类的实现方式都完全相同，如对于 PErrorStmt 和 PNoOpStmt 实现为单例类（因为这两种语句只有一种形式）、对于 PLabel 和 PTemp 类有一个全局的计数机制从而在翻译时方便的获取全局的 Label 标号和全局的 TEMP 号，以及 PTemp 还维护了前 20 个 Temp 的缓存池可以节省空间等。

**4.对 Minijava 的翻译**

对 Minijava 代码的翻译主要思路为：

（1）将 Minijava 每个类的每个方法翻译为 Piglet 的一个过程，其过程名为类名+_+方法名从而避免不同类的重名方法

（2）对于 if, while 等语句根据编译原理课讲述的翻译方案，翻译成用条件跳转语句 CJUMP 和 Label 构造的控制流。

（3）对于 new 语句新建对象或数组，根据前面所说的内存布局分配空间并进行初始化操作。

（4）使用引用型变量时均需要进行空指针检查，数组访问时需要进行下标越界检查。

（5）对于过程调用，如果超过 20 个参数，需要分配一块空间，将多于 19 个的参数放入这块空间，并将其地址作为第 20 个参数传入。

这部分翻译由一个 TranslateVisitor 的一次遍历完成，这个 Visitor 会负责语句的翻译，并包装成相应的 Piglet 代码类将其返回。具体语句翻译的注意点都在上面提到，因此翻译细节不再赘述。

附：一些关键代码的实现

Piglet 代码的抽象基类：

```java
public abstract class PCode {

    /**
     * 一级语句组的缩进宽度，建议大于4（否则三位数的label会占满4位缩进空间）
     */
    protected static final int TAB_WIDTH = 6;

    /**
     * 标记常量，用来表示该代码块会跨行
     */
    protected static final int MULTILINE = -1;

    /**
     * <p>对于单行代码段，直接打印该代码；
     * <p>对于多行代码段，在当前缩进宽度depth位置下输出该代码段。
     * @param depth 对于多行代码，当前代码块的缩进宽度
     * @return 对于单行代码，返回输出的字符数，对于多行代码，返回{@link PCode#MULTILINE}。
     */
    public abstract int print(int depth);

    /**
     * 工具方法，从当前位置往右缩进depth个空格
     * @param depth 缩进宽度
     */
    protected void indent(int depth) {
        for (int i = 0; i < depth; i++)
            System.out.print(' ');
    }

    /**
     * 工具方法，输出一个空格
     */
    protected void space() {
        System.out.print(' ');
    }

}
```

实现类举例：（以 CJUMP 为例）

```java
public class PCJumpStmt extends PStmt {

    private PExp pExp;

    private PLabel pLabel;

    public PCJumpStmt(PExp pExp, PLabel pLabel) {
        this.pExp = pExp;
        this.pLabel = pLabel;
    }

    @Override
    public int print(int depth) {
        System.out.print("CJUMP ");
        int l = pExp.print(depth + 6);
        // 若表达式为多行，则Label需要换行并与表达式开头对齐
        if (l == MULTILINE) {
            System.out.println();
            indent(depth + 6);
            pLabel.print(depth + 6);
        } else {
            space();
            pLabel.print(depth + 6 + l);
        }
        System.out.println();
        return MULTILINE;
    }

}
```

以下是一些语句的翻译代码举例。

数组取值（空引用检查，下标越界检查）：

```
public Object visit(ArrayLookup n, MIdentifier argu) {
    PStmtList stmtList = new PStmtList();
    PTemp array = PTemp.newTemp();
    PTemp len = PTemp.newTemp();
    PLabel ok1 = PLabel.newLabel();
    // Check array nonnull and get array length
    stmtList.add(new PMoveStmt(array, (PExp) n.f0.accept(this, argu)))
            .add(new PCJumpStmt(new PBinOp(PBinOpType.PLUS, array, new PInteger(1)), ok1))
            .add(PErrorStmt.getInstance())
            .add(ok1)
            .add(new PLoadStmt(len, array, 0));
    // Check 0<=index<length and get value
    PTemp index = PTemp.newTemp();
    PTemp value = PTemp.newTemp();
    PLabel error = PLabel.newLabel();
    PLabel ok2 = PLabel.newLabel();
    PLabel ok3 = PLabel.newLabel();
    stmtList.add(new PMoveStmt(index, (PExp) n.f2.accept(this, argu)))
            .add(new PCJumpStmt(new PBinOp(PBinOpType.LT, index, new PInteger(0)), ok2))
            .add(PErrorStmt.getInstance())
            .add(ok2)
            .add(new PCJumpStmt(new PBinOp(PBinOpType.LT, index, len), error))
            .add(new PJumpStmt(ok3))
            .add(error)
            .add(PErrorStmt.getInstance())
            .add(ok3)
            .add(new PLoadStmt(value,
                    new PBinOp(PBinOpType.PLUS, array, new PBinOp(PBinOpType.TIMES, index, new PInteger(4))), 4));
    return new PStmtExp(stmtList, value);
}
```

while 语句（简单控制流）：

```
public Object visit(WhileStatement n, MIdentifier argu) {
    ArrayList<PStmt> stmts = new ArrayList<>();
    PLabel start = PLabel.newLabel();
    PLabel end = PLabel.newLabel();
    stmts.add(start);
    stmts.add(new PCJumpStmt((PExp) n.f2.accept(this, argu), end));
    Object obj = n.f4.accept(this, argu);
    if (obj instanceof PStmt)
        stmts.add((PStmt) obj);
    else
        stmts.addAll((ArrayList<PStmt>) obj);
    stmts.add(new PJumpStmt(start));
    stmts.add(end);
    stmts.add(PNoOpStmt.getInstance());
    return stmts;
}
```

&&表达式（短路与的控制流）：

```
public Object visit(AndExpression n, MIdentifier argu) {
    PStmtList stmtList = new PStmtList();
    PTemp tmp = PTemp.newTemp();
    PLabel label = PLabel.newLabel();
    stmtList.add(new PMoveStmt(tmp, new PInteger(0)))
            .add(new PCJumpStmt((PExp) n.f0.accept(this, argu), label))
            .add(new PCJumpStmt((PExp) n.f2.accept(this, argu), label))
            .add(new PMoveStmt(tmp, new PInteger(1)))
            .add(label)
            .add(PNoOpStmt.getInstance());
    return new PStmtExp(stmtList, tmp);
}
```

方法调用语句（空引用检查，超过 20 个参数的封装）：

```
public Object visit(MessageSend n, MIdentifier argu) {
    PStmtList stmtList = new PStmtList();
    PTemp dTable = PTemp.newTemp();
    PTemp func = PTemp.newTemp();
    PTemp obj = PTemp.newTemp();
    PLabel ok = PLabel.newLabel();
    PExp exp = (PExp) n.f0.accept(this, argu);
    MMethod method = (MMethod) n.f2.accept(this, (MClass) exp.getType());
    int id = method.getId();
    // Check obj nonnull and get method address
    stmtList.add(new PMoveStmt(obj, exp))
            .add(new PCJumpStmt(new PBinOp(PBinOpType.PLUS, obj, new PInteger(1)), ok))
            .add(PErrorStmt.getInstance())
            .add(ok)
            .add(new PLoadStmt(dTable, obj, 0))
            .add(new PLoadStmt(func, dTable, id * 4));
    PCall call = new PCall(new PStmtExp(stmtList, func));
    // If param cnt exceeds 19, allocate space and pass pointer by the 20th param
    ArrayList<Object> paramList = (ArrayList<Object>) n.f4.accept(this, argu);
    if (paramList == null)
        paramList = new ArrayList<>();
    call.addParam(obj);
    int l = paramList.size();
    if (l < 20)
        for (Object param : paramList)
            call.addParam((PExp) param);
    else {
        for (int i = 0; i < 18; i++)
            call.addParam((PExp) paramList.get(i));
        PTemp extra = PTemp.newTemp();
        PStmtList stmts = new PStmtList();
        stmts.add(new PMoveStmt(extra, new PAllocate(new PInteger((l - 18) * 4))));
        for (int i = 18; i < l; i++)
            stmts.add(new PStoreStmt(extra, (i - 18) * 4, (PExp) paramList.get(i)));
        call.addParam(new PStmtExp(stmts, extra));
    }
    return call.setType(method.getReturnType());
```

新建类实例的 new 语句（初始化方法表、属性表，循环将属性表清零）：

```
public PStmtExp getNewPExp() {
    PStmtList stmtList = new PStmtList();
    PTemp dTable = PTemp.newTemp();
    PTemp vTable = PTemp.newTemp();
    int l;
    // Initialize dTable
    l = methodList.size();
    if (l > 0) {
        stmtList.add(new PMoveStmt(dTable, new PAllocate(new PInteger(l * 4))));
        for (MMethod method : methodList) {
            stmtList.add(new PStoreStmt(dTable, method.getId() * 4, new PIdentifier(method.getFullName())));
        }
    } else
        stmtList.add(new PMoveStmt(dTable, new PInteger(0)));
    // Initialize vTable
    l = fields.size();
    stmtList.add(new PMoveStmt(vTable, new PAllocate(new PInteger(4 + l * 4))));
    if (l > 0) {
        PTemp loop = PTemp.newTemp();
        PLabel start = PLabel.newLabel();
        PLabel end = PLabel.newLabel();
        // Loop to memset vTable to 0
        stmtList.add(new PMoveStmt(loop, new PInteger(4))).add(start)
                .add(new PCJumpStmt(new PBinOp(PBinOpType.LT, loop, new PInteger(4 + 4 * l)), end))
                .add(new PStoreStmt(new PBinOp(PBinOpType.PLUS, vTable, loop), 0, new PInteger(0)))
                .add(new PMoveStmt(loop, new PBinOp(PBinOpType.PLUS, loop, new PInteger(4))))
                .add(new PJumpStmt(start))
                .add(end);
    }
    stmtList.add(new PStoreStmt(vTable, 0, dTable));
    return (PStmtExp) new PStmtExp(stmtList, vTable).setType(this);
}
```

新建数组的 new 语句（检查数组长度非负，写入数组长度，循环将数组清零）：

```
public Object visit(ArrayAllocationExpression n, MIdentifier argu) {
    PStmtList stmtList = new PStmtList();
    PTemp len = PTemp.newTemp();
    PTemp array = PTemp.newTemp();
    // len = exp
    stmtList.add(new PMoveStmt(len, (PExp) n.f3.accept(this, argu)));
    // Check len >= 0 and allocate space, otherwise error
    PLabel error = PLabel.newLabel();
    stmtList.add(new PCJumpStmt(new PBinOp(PBinOpType.LT, len, new PInteger(0)), error))
            .add(PErrorStmt.getInstance())
            .add(error)
            .add(new PMoveStmt(array, new PAllocate(new PBinOp(PBinOpType.TIMES,
                    new PBinOp(PBinOpType.PLUS, len, new PInteger(1)), new PInteger(4)))));
    // Loop to memset array to 0
    PTemp loop = PTemp.newTemp();
    PLabel start = PLabel.newLabel();
    PLabel end = PLabel.newLabel();
    stmtList.add(new PMoveStmt(loop, new PInteger(4)))
            .add(start)
            .add(new PCJumpStmt(new PBinOp(PBinOpType.LT, loop, new PBinOp(PBinOpType.TIMES,
                    new PBinOp(PBinOpType.PLUS, len, new PInteger(1)), new PInteger(4))), end))
            .add(new PStoreStmt(new PBinOp(PBinOpType.PLUS, array, loop), 0, new PInteger(0)))
            .add(new PMoveStmt(loop, new PBinOp(PBinOpType.PLUS, loop, new PInteger(4))))
            .add(new PJumpStmt(start))
            .add(end)
            .add(new PStoreStmt(array, 0, len));
    return new PStmtExp(stmtList, array);
}
```

# 三、从 Piglet 到 SPiglet

编译器的下一步是将 Piglet 代码翻译成另一种中间语言 SPiglet。与 Piglet 相比，SPiglet 仅仅去掉了嵌套表达式的限制，使得语句都成为了类似三地址代码的形式，更加接近最终的 MIPS 形式。这一步翻译思路十分简单，就是对于之前 Piglet 中语句可以使用 Exp，而如今只能使用 Temp 或者 SimpleExp 的位置，将原本的 Exp 移动到新的 TEMP 中即可，相当于使用一些新的临时单元将原来的嵌套表达式一层层拆开。

思路：在这一步中使用两个 Visitor，第一个 MaxTempVisitor 负责遍历语法树找到原本 Piglet 程序用到的最大的 Temp 号，从而从这里往后分配 TEMP 号，由于不需要往下传递信息，但需要往上返回用到的 Temp 号，因此使用 GJNoArguDepthFirst 基类 Visitor。第二个 TranslateVisitor 负责进行语句的翻译，对于语句中的表达式，将该句型中需要的表达式类型往下传，如果表达式合法则直接返回原表达式，否则使用 MOVE 语句将结果存入新的临时单元，并将新临时单元返回。

由于 SPiglet 不存在很复杂的缩进模式，因此直接打印即可，不需要像上一次 Piglet 代码一样包装起来再打印。对于 TranslateVisitor，由于对于表达式不确定打印的是什么（可能是原表达式或新建的临时单元），因此需要返回一个代码包装，建立 SCode 类进行这项操作，其中只包装一个 String。另外使用 STemp 类维护全局的 Temp 号。

附：一些关键代码的实现

临时单元的分配类 STemp：

```
public class STemp extends SCode {

    private static int TempCnt;

    /**
     * 初始化全局当前寄存器号
     * @param total 当前已用寄存器
     */
    public static void init(int total) {
        TempCnt = total;
    }

    public static STemp newTemp() {
        return new STemp(++TempCnt);
    }

    private int id;

    public STemp(int id) {
        this.id = id;
    }

    @Override
    public String toString() {
        return String.format("TEMP %d", id);
    }

}
```

语句的翻译（举例）：

```
public SCode visit(HLoadStmt n, ExpType argu) {
    SCode exp1 = n.f1.accept(this, ExpType.TEMP);
    SCode exp2 = n.f2.accept(this, ExpType.TEMP);
    System.out.println(String.format("\tHLOAD %s %s %s", exp1, exp2, n.f3.f0.tokenImage));
    return null;
}

/**
 * f0 -> "MOVE"
 * f1 -> Temp()
 * f2 -> Exp()
 */
public SCode visit(MoveStmt n, ExpType argu) {
    SCode exp1 = n.f1.accept(this, ExpType.TEMP);
    SCode exp2 = n.f2.accept(this, ExpType.EXP);
    System.out.println(String.format("\tMOVE %s %s", exp1, exp2));
    return null;
}

/**
 * f0 -> "PRINT"
 * f1 -> Exp()
 */
public SCode visit(PrintStmt n, ExpType argu) {
    SCode exp = n.f1.accept(this, ExpType.SIMPLEEXP);
    System.out.println(String.format("\tPRINT %s", exp));
    return null;
}
```

表达式的翻译（举例）：

```
public SCode visit(HAllocate n, ExpType argu) {
    SCode exp = n.f1.accept(this, ExpType.SIMPLEEXP);
    String s = String.format("HALLOCATE %s", exp);

    if (argu == ExpType.EXP)
        return new SCode(s);

    SCode ret = STemp.newTemp();
    System.out.println(String.format("\tMOVE %s %s", ret, exp));
    return ret;
}

/**
 * f0 -> Operator()
 * f1 -> Exp()
 * f2 -> Exp()
 */
public SCode visit(BinOp n, ExpType argu) {
    SCode exp1 = n.f1.accept(this, ExpType.TEMP);
    SCode exp2 = n.f2.accept(this, ExpType.SIMPLEEXP);
    String s = String.format("%s %s %s", ((NodeToken) n.f0.f0.choice).tokenImage, exp1, exp2);

    if (argu == ExpType.EXP)
        return new SCode(s);

    SCode ret = STemp.newTemp();
    System.out.println(String.format("\tMOVE %s %s", ret, s));
    return ret;
}
```

与 ucla 官网上的样例相比，本程序翻译出的 SPiglet 代码能够使用更少的临时变量。因为 ucla 样例程序的翻译是不管需要 SimpleExp 还是 Temp，都直接将原表达式 MOVE 到 Temp 中，然后使用新的 Temp，但对于一些情况比如原本表达式是一个数字或者 Label，现在需要一个 SimpleExp，就可以直接使用数字或 Label 而不需要将其移动到一个新的临时变量单元中。下图为 ucla 样例（LinearSearch.spg）与本程序输出（对 LinearSearch.pg 的翻译结果）的对比（前者共使用 161 个 Temp，后者仅使用 139 个）：

```
MAIN
        MOVE TEMP 36 HALLOCATE 16
        MOVE TEMP 37 HALLOCATE 12
        MOVE TEMP 87 LS_Init
        HSTORE TEMP 36 12 TEMP 87
        MOVE TEMP 88 LS_Search
        HSTORE TEMP 36 8 TEMP 88
        MOVE TEMP 89 LS_Print
        HSTORE TEMP 36 4 TEMP 89
        MOVE TEMP 90 LS_Start
        HSTORE TEMP 36 0 TEMP 90
        MOVE TEMP 38 4
L0      NOOP
        MOVE TEMP 91 12
        MOVE TEMP 92 LT TEMP 38 TEMP 91
        CJUMP TEMP 92 L1
        MOVE TEMP 93 PLUS TEMP 37 TEMP 38
        MOVE TEMP 94 0
        HSTORE TEMP 93 0 TEMP 94
        MOVE TEMP 38 PLUS TEMP 38 4
        JUMP L0
L1      NOOP
        HSTORE TEMP 37 0 TEMP 36
        MOVE TEMP 35 TEMP 37
        HLOAD TEMP 33 TEMP 35 0
        HLOAD TEMP 34 TEMP 33 0
        MOVE TEMP 95 10
        MOVE TEMP 96 CALL TEMP 34( TEMP 35 TEMP 95 )
        PRINT TEMP 96
END
```

```
MAIN
        MOVE TEMP 36 HALLOCATE 16
        MOVE TEMP 37 HALLOCATE 12
        MOVE TEMP 87 LS_Init
        HSTORE TEMP 36 12 TEMP 87
        MOVE TEMP 88 LS_Search
        HSTORE TEMP 36 8 TEMP 88
        MOVE TEMP 89 LS_Print
        HSTORE TEMP 36 4 TEMP 89
        MOVE TEMP 90 LS_Start
        HSTORE TEMP 36 0 TEMP 90
        MOVE TEMP 38 4
L0      MOVE TEMP 91 LT TEMP 38 12
        CJUMP TEMP 91 L1
        MOVE TEMP 92 PLUS TEMP 37 TEMP 38
        MOVE TEMP 93 0
        HSTORE TEMP 92 0 TEMP 93
        MOVE TEMP 38 PLUS TEMP 38 4
        JUMP L0
L1      HSTORE TEMP 37 0 TEMP 36
        MOVE TEMP 35 TEMP 37
        HLOAD TEMP 33 TEMP 35 0
        HLOAD TEMP 34 TEMP 33 0
        MOVE TEMP 94 10
        MOVE TEMP 95 CALL TEMP 34 (TEMP 35 TEMP 94)
        PRINT TEMP 95
END
```

# 四、从 SPiglet 到 Kanga

在得到了 SPiglet 代码后，下一步是将其翻译成 Kanga 中间语言。**Kanga 与 SPiglet 的区别主要有：**

（1）标号从局部的变成全局的（虽然在我们之前的实现中，使用的标号的确是全局的）

（2）寄存器的数量从原来的无限个变成现在的 24 个，其中 a0~a3 存放函数调用的参数，v0 用于存放函数返回值，v1 备用，s0~s7 为 caller-saved 寄存器，t0~t9 为 callee-saved 寄存器。

（3）引入了运行栈的概念，可以使用指令从栈中加载和往栈上储存值。Kanga 中的栈是针对每个过程本身而言的一帧。

（4）函数调用时多余的参数通过专门的指令往被调用函数的栈上传递。

这次翻译主要是进行寄存器的分配工作，主要思路为：对于 SPiglet 的每个过程，将每个语句看做一个节点建立程序流图，初始化每个语句的 def 和 use 集合，然后对该程序流图进行活跃变量的数据流分析，从而得到每个临时变量的活跃区间，再使用线性扫描算法进行寄存器的分配。

**关于寄存器分配的思路和注意点如下：**

（1）对于死语句（即对变量赋值但赋值从未使用的语句）而言，它的存在会影响活跃区间的数据流分析，一种解决方法是消除死语句，但这样比较麻烦；另一种方法是修改活跃区间的定义，定义为一个变量从第一次出现到最后一次被使用的区间，这样一定程度上会扩大变量的活跃区间从而造成寄存器分配效率有所降低，但对于前一步生成的 SPiglet 代码而言，不会出现一个临时变量被多次复用的情况，因此不会对寄存器分配效率产生区别。本程序采用后一种解决办法。

（2）对于跨越函数调用的语句，如果一个变量被保存在 caller-saved 寄存器中，那么在函数调用前需要将其压栈保存，函数调用后需要恢复其值；与此同时，在每个函数开头和结尾需要将 callee-saved 寄存器进行保存和恢复，这就产生了四个需要保存和恢复寄存器的时机。为了方便起见我们采取这样的措施（也是 ucla 样例的翻译采用的措施）：如果一个临时

变量的活跃区间跨越了某次函数调用，那么就不将其分配到 caller-saved 寄存器中。这样反正 callee-saved 寄存器在函数开头结尾被保存和恢复，就不用处理 caller-saved 寄存器的保存与恢复了。

（3）对于每个临时变量，我们要么将其分配到一个 caller-saved 寄存器中，要么将其分配到一个 callee-saved 寄存器中，要么将其溢出，在每次使用时从栈上加载，在每次赋值后写回栈上，不会出现一个临时变量的活跃区间分成两段，两段分别分配不同的寄存器的情况（为了方便起见）。

（4）关于局部 Label 的处理，对每个 Label 将其名称扩展为方法名+_+Label 名，然后使用一个全局的 Label 管理类将每个 Label 映射为一个全局的标号，在翻译时将 Label 翻译成对应标号的 Label。

以下是具体翻译过程：

**1.关于程序流图和活性区间的模型：**

**Statement 类表示程序流图中的一个节点**，也是一个 SPiglet 语句，它包含六个集合：

prev, next 是该节点的前驱和后继节点集合；

def, use 是该节点本身定义和使用的变量；

in, out 是该节点执行前后的活跃变量，在数据流分析时进行更新。

**Method 类表示一个 SPiglet 过程**，主要包括以下字段：

name：过程名；

param, stack, callParam：过程头部的三个参数，即参数个数、使用栈单元个数和最大调用参数；

statements：程序流图节点的列表；

callPos：过程中所有调用语句的位置集合，用来决定一个活性区间是否跨越函数调用从而应该分配为 callee-saved 寄存器；

**Interval 类表示一个活性区间**，包括以下字段：

begin, end：区间开始和结束位置；

acrossCall：是否跨越函数调用；

tempId：该区间对应的临时单元号。

**2.建立程序流图**

本步骤使用了两个 Visitor 进行遍历。

第一个 BuildSymbolTableVisitor 对每个过程建立一个 Method，并将该过程的每个语句添加进 Method 中，相当于明确了每个过程的程序流图的节点，这里需要对每个过程额外添加一个入口语句和一个出口语句。对于 Call 语句而言，它将当前的语句位置添加到 Method 中维护。同时对于每个 Label，它生成相应的 Label 扩展名并将该 Label 与其位置添加到 Label 全局管理类中。

第二个 BuildFlowGraphVisitor 建立程序流图。对于每一个过程中的每一个语句，更新其前驱和后缀信息。对于一般的语句而言，其后缀就是后面一条语句；对于 Jump 语句而言后缀是它所跳转的 Label 的语句位置；对于 CJump 语句而言会有两个后缀：后面一条语句和它所跳转的 Label 的语句位置。

**3.进行活性分析和寄存器分配**

本步骤在符号表内部进行，对于每一个过程，先后对其进行活性分析、活性区间的确定和线性扫描三个步骤。

（1）活性分析：对整个程序流图反复进行，对每个节点将其 in 集合更新为(out-def) ∩use，out 集合更新为所有后继 in 集合的并集，直到一轮更新中没有节点的 in 发生改变为止。

（2）确定每个变量的活性区间：将每个变量的活性区间设置为从该变量第一次出现到该变量最后一次出现在 in 集合中。对于参数而言，认为其第一次出现在一个过程的开头。同时，判断活性区间是否跨越某个函数调用，若是则设置其标志位。

（3）使用线性扫描算法进行寄存器分配。对于每个 caller-saved 和 callee-saved 寄存器，记录当前占用它的活性区间。对所有活性区间按照起点顺序进行排序，从前往后处理每一个区间：

对于一个区间而言，先检查当前每个寄存器是否有已经结束的区间（即完全在当前区间之前），如果有将该寄存器释放，将该寄存器分配给那个区间对应的变量；

然后，除非该区间跨越函数调用，否则先尝试分配 caller-saved 寄存器，找最小的空闲的 caller-saved 寄存器，如果能找到就将该区间填入，结束；否则找到所有 caller-saved 寄存器中最晚结束的区间，若早于该区间则互换（为了保证占用寄存器的一定是最晚结束的区间）；

接下来尝试分配 callee-saved 寄存器，找最小的空闲的 callee-saved 寄存器，能找到就填入，结束；否则找到所有 callee-saved 寄存器中最晚结束的区间，若早于该区间则互换（为了保证占用寄存器的一定是最晚结束的区间）；

最后将该区间对应的变量溢出（实在找不到寄存器可用）。

处理完所有区间后，为当前占用各个寄存器的最后几个区间对应的变量分配这些寄存器。然后统一处理溢出的变量，为每个溢出的变量分配一个栈单元。

**4.将 SPiglet 翻译为 Kanga 代码**

本步骤使用一个 TranslateVisitor 进行翻译。该 Visitor 主要完成以下任务：

（1）将所有 SPiglet 语句中的 Temp 换成分配的寄存器。对于溢出的情况，将该 Temp 的值从栈载入到 v0 或 v1 这两个备用寄存器中，然后使用备用寄存器。

（2）在每个过程的开头保存所有用到的 callee-saved 寄存器，然后载入所有参数，在过程结尾恢复所有用到的 callee-saved 寄存器。

（3）对于 call 语句，将前四个参数放进 a0~a3 寄存器，将多于四个的参数使用 PASSARG 传递。将返回值从 a0 移出。

（4）对于 Label，使用全局重新分配的 Label 进行替换输出。

附：一些关键代码的实现

过程内部的活性分析：

```java
/**
 * 进行活性分析，不断迭代直到活性区间收敛
 */
public void activityAnalyze() {
    boolean done = false;
    while (!done) {
        done = true;
        for (int id = statements.size() - 1; id >= 0; id--) {
            Statement statement = statements.get(id);
            if (statement.activityAnalyze())
                done = false;
        }
    }
}
```

```java
/**
 * 更新该节点的活跃变量集合
 * @return 是否有变化
 */
public boolean activityAnalyze() {
    for (Statement next : next)
        out.addAll(next.in);
    HashSet<Integer> newIn = new HashSet<>();
    newIn.addAll(out);
    newIn.removeAll(def);
    newIn.addAll(use);
    if (!in.equals(newIn)) {
        in = newIn;
        return true;
    }
    return false;
}
```

计算变量的活跃区间：

```java
/**
 * 计算所有变量的活性区间
 */
public void computeAllInteval() {
    for (int id = 0; id < statements.size(); id++) {
        Statement statement = statements.get(id);
        for (Integer temp : statement.in)
            tempInterval.get(temp).end = id;
    }
    for (Interval interval : tempInterval.values())
        for (int callPos : callPos)
            if (interval.begin < callPos && interval.end > callPos)
                interval.acrossCall = true;
}
```

线性扫描算法：

```
public void allocateRegisters() {
    ArrayList<Interval> intervals = new ArrayList<>();
    intervals.addAll(tempInterval.values());
    Collections.sort(intervals);

    // Current interval of each register
    Interval[] TInterval = new Interval[10];
    Interval[] SInterval = new Interval[8];

    for (Interval interval : intervals) {
        int lastT = -1, lastS = -1, emptyT = -1, emptyS = -1;
        // Check t0~t9 for available regT and last regT
        for (int t = 9; t >= 0; t--)
            if (TInterval[t] != null) {
                if (TInterval[t].end <= interval.begin) {
                    // This interval has ended, record it and clear the register.
                    regT.put(TInterval[t].tempId, "t" + t);
                    TInterval[t] = null;
                    emptyT = t;
                } else if (lastT == -1 || TInterval[t].end > TInterval[lastT].end)
                    lastT = t;
            } else
                emptyT = t;
        // Check s0~s7 for available regS and last regS
        for (int s = 7; s >= 0; s--)
            if (SInterval[s] != null) {
                if (SInterval[s].end <= interval.begin) {
                    // This interval has ended, record it and clear the register.
                    regS.put(SInterval[s].tempId, "s" + s);
                    maxS = Math.max(maxS, s);
                    SInterval[s] = null;
                    emptyS = s;
                } else if (lastS == -1 || SInterval[s].end > SInterval[lastS].end)
                    lastS = s;
            } else
                emptyS = s;
```

```
        // Assign reg T
        if (!interval.acrossCall)
            if (emptyT != -1) {
                // put it in empty T
                TInterval[emptyT] = interval;
                interval = null;
            } else if (interval.end < TInterval[lastT].end) {
                // had better use the register for shorter interval
                Interval tmp = TInterval[lastT];
                TInterval[lastT] = interval;
                interval = tmp;
            }
        // Assign reg S
        if (interval != null)
            if (emptyS != -1) {
                // put it in empty S
                SInterval[emptyS] = interval;
                interval = null;
            } else if (interval.end < SInterval[lastS].end) {
                // had better use the register for shorter interval
                Interval tmp = SInterval[lastS];
                SInterval[lastS] = interval;
                interval = tmp;
            }
        // No T and S available any more, spill it
        if (interval != null)
            regSpilled.put(interval.tempId, null);
    }
    for (int t = 0; t < 10; t++)
        if (TInterval[t] != null)
            regT.put(TInterval[t].tempId, "t" + t);
    for (int s = 0; s < 8; s++)
        if (SInterval[s] != null) {
            regS.put(SInterval[s].tempId, "s" + s);
            maxS = Math.max(maxS, s);
        }

    // Stack units has 3 parts:
    // 1.params more than 4
    // 2.callee-saved S
    // 3.spilled regs
    int stackId = Math.max(param - 4, 0) + maxS;
    for (Integer tempId : regSpilled.keySet())
        regSpilled.put(tempId, "SPILLEDARG " + stackId++);
    stack = stackId;
}
```

标号的全局管理：

```
public class Label {

    private static int labelCnt = 0;

    private static HashMap<String, Integer> labelToId = new HashMap<>();
    private static HashMap<String, Integer> labelToName = new HashMap<>();

    /**
     * 将一个方法局部标号与语句序号相联系
     * @param label 标号
     * @param id 所在语句的序号
     */
    public static void add(String label, int id) {
        labelToId.put(label, id);
        labelToName.put(label, labelCnt++);
    }

    /**
     * 获取一个标号所在的语句序号
     * @param label 标号
     * @return 语句序号
     */
    public static int getId(String label) {
        return labelToId.get(label);
    }

    /**
     * 获取一个局部标号的全局名称
     * @param label 标号
     * @return 全局序号
     */
    public static int getName(String label) {
        return labelToName.get(label);
    }

}
```

过程开头结尾保存 callee-saved 寄存器以及载入参数：

```
/**
 * 在方法开头保存所有的callee-saved寄存器
 */
public void storeReg() {
    int stackId = Math.max(param - 4, 0);
    for (int i = 0; i < maxS; i++)
        System.out.println(String.format("\tASTORE SPILLEDARG %d s%d", i + stackId, i));
}

/**
 * 在方法结尾恢复所有的callee-saved寄存器
 */
public void restoreReg() {
    int stackId = Math.max(param - 4, 0);
    for (int i = 0; i < maxS; i++)
        System.out.println(String.format("\tALOAD s%d SPILLEDARG %d", i, i + stackId));
}

/**
 * 在方法开头加载所有参数
 */
public void loadParam() {
    int i;
    for (i = 0; i < param && i < 4; i++)
        if (tempInterval.containsKey(i))
            putReg(i, "a" + i);
    for (; i < param; i++)
        if (tempInterval.containsKey(i))
            if (regSpilled.containsKey(i)) {
                System.out.println(String.format("\tALOAD v0 SPILLEDARG %d", i - 4));
                putReg(i, "v0");
            } else
                System.out.println(String.format("\tALOAD %s SPILLEDARG %d", getReg(i, null), i - 4));
}
```

```
public Object visit(StmtExp n, Method method) {
    method.storeReg();
    method.loadParam();
    n.f1.accept(this, method);
    System.out.println("\tMOVE v0 " + n.f3.accept(this, method));
    method.restoreReg();
    return null;
}
```

函数调用语句的翻译：

```
/**
 * f0 -> "CALL"
 * f1 -> SimpleExp()
 * f2 -> "("
 * f3 -> ( Temp() )*
 * f4 -> ")"
 */
public Object visit(Call n, Method method) {
    Vector<Node> paramNode = n.f3.nodes;
    int l = paramNode.size();
    int i;
    for (i = 0; i < l && i < 4; i++)
        System.out.println(String.format("\tMOVE a%d %s", i,
                method.getReg((Integer) paramNode.get(i).accept(this, method), "v0")));
    for (; i < l; i++)
        System.out.println(String.format("\tPASSARG %d %s", i - 3,
                method.getReg((Integer) paramNode.get(i).accept(this, method), "v0")));
    System.out.println("\tCALL " + n.f1.accept(this, method));
    return "v0";
}
```

将 Temp 替换为寄存器的通用例程：

```
/**
 * 从一个变量中加载值
 * @param tempId 变量id
 * @param regTemp 可能需要的临时寄存器
 * @return 值所在的寄存器
 */
public String getReg(int tempId, String regTemp) {
    if (regT.containsKey(tempId))
        return regT.get(tempId);
    else if (regS.containsKey(tempId))
        return regS.get(tempId);
    else {
        System.out.println(String.format("\tALOAD %s %s", regTemp, regSpilled.get(tempId)));
        return regTemp;
    }
}

/**
 * 往一个变量中保存值
 * @param tempId 变量id
 * @param exp 需要保存的表达式
 */
public void putReg(int tempId, String exp) {
    if (regSpilled.containsKey(tempId)) {
        if (!exp.equals("v0"))
            System.out.println(String.format("\tMOVE v0 %s", exp));
        System.out.println(String.format("\tASTORE %s v0", regSpilled.get(tempId)));
    } else {
        String r = getReg(tempId, null);
        if (!r.equals(exp))
            System.out.println(String.format("\tMOVE %s %s", r, exp));
    }
}
```

本程序寄存器分配的结果比 ucla 官网上的样例的分配方案更优（使用更少的栈单元），下图为对比（左图为 ucla 样例的 BinaryTree.kg，右图为本程序对 BinaryTree.spg 的翻译结果）：左图使用 10 个栈单元，右图仅使用 8 个；画圈的位置为一个显著差异：同一个语句中本翻译方案会尝试使用同一个寄存器。

```
Tree_Delete [2][10][3]
        ASTORE SPILLEDARG 0 s0
        ASTORE SPILLEDARG 1 s1
        ASTORE SPILLEDARG 2 s2
        ASTORE SPILLEDARG 3 s3
        ASTORE SPILLEDARG 4 s4
        ASTORE SPILLEDARG 6 s6
        ASTORE SPILLEDARG 7 s7
        ASTORE SPILLEDARG 8 s5
        MOVE s0 a0
        MOVE s1 a1
        MOVE s2 s0
        MOVE s3 s0
        MOVE s4 1
        MOVE v1 0
        ASTORE SPILLEDARG 5 v1
        MOVE s6 1
L18     NOOP
        CJUMP s4 L19
        MOVE t0 s2
        HLOAD t1 t0 0
        HLOAD t2 t1 20
        MOVE a0 t0
        CALL t2
        MOVE t1 v0
        MOVE s7 t1
        MOVE t0 LT s1 s7
        CJUMP t0 L20
```

```
Tree_Delete[2][8][3]
    ASTORE SPILLEDARG 0 s0
    ASTORE SPILLEDARG 1 s1
    ASTORE SPILLEDARG 2 s2
    ASTORE SPILLEDARG 3 s3
    ASTORE SPILLEDARG 4 s4
    ASTORE SPILLEDARG 5 s5
    ASTORE SPILLEDARG 6 s6
    MOVE s0 a0
    MOVE s1 a1
    MOVE s2 s0
    MOVE s3 s0
    MOVE s4 1
    MOVE v0 0
    ASTORE SPILLEDARG 7 v0
    MOVE s6 1
L16 NOOP
    CJUMP s4 L30
    MOVE t0 s2
    HLOAD t1 t0 0
    HLOAD t1 t1 20
    MOVE a0 t0
    CALL t1
    MOVE t0 v0
    MOVE s7 t0
    MOVE t0 LT s1 s7
    CJUMP t0 L19
```

# 五、从 Kanga 到 MIPS

在得到了 Kanga 的有限寄存器、三地址代码后，离 MIPS 只有最后一步了。这一步要做的是将 Kanga 的各语句翻译成对等的 MIPS 指令，同时需要对运行栈进行实际的维护。在 Kanga 的语法中，每个函数有自己的一段空间作为运行栈，而在 MIPS 中，需要在每个函数的开头和结尾处对当前栈帧进行维护。同时，对于 Kanga 中的 SPILLEDARG 和 PASSARG 需要到栈上实际进行定位，可以说这是本次翻译的唯一任务。

翻译的思路如下：

（1）对于与系统调用相关的指令如 HALLOCATE、PRINT 和 ERROR，将系统调用包装成一个小函数从而方便的生成代码。

（2）对于每个函数，建立这样的栈帧：

| （上一函数的栈帧） |
| --- |
| 返回地址（4 字节） |
| 上一个栈帧的开始地址（4 字节） |
| 本函数实际使用的栈单元（指定的栈单元-超过四个参数占用的栈单元） |
| 传给下一个栈帧的参数（最大调用参数超过四个的部分） |
| （下一函数的栈帧） |

附：一些关键代码的实现

函数栈帧的计算：

```java
public class Method {

    private int extraParam, stack, maxcall, pos;

    public Method(int param, int stack, int maxcall) {
        this.extraParam = Math.max(param - 4, 0);
        this.stack = stack;
        this.maxcall = maxcall;
        pos = Math.max(maxcall - 4, 0);
    }

    /**
     * 获取本过程实际的栈帧大小
     * @return 实际栈帧大小
     */
    public int getActual() {
        return (2 + stack - extraParam + pos) * 4;
    }

    /**
     * 将对SpilledReg的访问转为栈地址的访问，如果是本程序的参数，去上一个栈帧找；如果是本程序的栈单元，到中间一段找。
     * @param id 栈单元id
     * @return 栈地址
     */
    public String getSpilledReg(int id) {
        if (id < extraParam)
            return String.format("%d($fp)", id * 4);
        return String.format("%d($sp)", (id - extraParam + pos) * 4);
    }
}
```

函数栈帧的维护：

```java
public void visit(Procedure n, Method argu) {
    // Declaration
    String name = n.f0.f0.tokenImage;
    System.out.println("\n\t.text");
    System.out.println(String.format("\t.globl\t%s", name));
    System.out.println(String.format("%s:", name));
    // Build stack
    int param = Integer.parseInt(n.f2.f0.tokenImage);
    int stack = Integer.parseInt(n.f5.f0.tokenImage);
    int maxcall = Integer.parseInt(n.f8.f0.tokenImage);
    Method method = new Method(param, stack, maxcall);
    int actual = method.getActual();
    System.out.println("\tsw $fp, -8($sp)");
    System.out.println("\tmove $fp, $sp");
    System.out.println(String.format("\tsubu $sp, $sp, %d", actual));
    System.out.println("\tsw $ra, -4($fp)");
    // Visit statements
    n.f10.accept(this, method);
    // Dealloc stack
    System.out.println("\tlw $ra, -4($fp)");
    System.out.println(String.format("\tlw $fp, %d($sp)", actual - 8));
    System.out.println(String.format("\taddu $sp, $sp, %d", actual));
    System.out.println("\tj $ra");
}
```

系统调用例程的包装（附在每个程序后面的 MIPS 代码）：

```
        .text
        .globl _halloc
_halloc:
        li $v0, 9
        syscall
        j $ra

        .text
        .globl _print
_print:
        li $v0, 1
        syscall
        la $a0, newl
        li $v0, 4
        syscall
        j $ra

        .text
        .globl _error
_error:
        la $a0, str_er
        li $v0, 4
        syscall
        li $v0, 10
        syscall
        j $ra

        .data
        .align  0
newl:   .asciiz "\n"

        .data
        .align  0
str_er: .asciiz " ERROR: abnormal termination\n"
```

此外，本程序生成的 MIPS 代码中，函数所分配的栈帧也比 ucla 官网上样例分配的栈帧要更小，下图为对比（左图为 ucla 官网上的 MoreThan4.s，右图为本程序对于 MoreThan4.kg 的翻译结果）：

```
        .text
        .globl          MT4_Start
MT4_Start:
        sw $fp, -8($sp)
        move $fp, $sp
        subu $sp, $sp, 48
        sw $ra, -4($fp)
        sw $s0, 24($sp)
        sw $s1, 28($sp)
        sw $s2, 32($sp)
        sw $s3, 36($sp)
        move $s0 $a0
        move $s1 $a1
        move $s2 $a2
        move $s3 $a3
        move $a0 $s1
        jal _print
        move $a0 $s2
        jal _print
        move $a0 $s3
```

```
        .text
        .globl  MT4_Start
MT4_Start:
        sw $fp, -8($sp)
        move $fp, $sp
        subu $sp, $sp, 36
        sw $ra, -4($fp)
        sw $s0, 12($sp)
        sw $s1, 16($sp)
        sw $s2, 20($sp)
        sw $s3, 24($sp)
        move $s0 $a0
        move $s1 $a1
        move $s2 $a2
        move $s3 $a3
        move $a0 $s1
        jal _print
        move $a0 $s2
        jal _print
        move $a0 $s3
        jal _print
        lw $v1, 0($fp)
```