编译原理实习报告

王子辰*

北京大学

2017 年 1 月

*电子邮件: wzc1995@pku.edu.cn 学号: 1400012930

目录

1 写在前面

1.1 目标

本学期编译实习的课程要求是写一个简化的 Java 编译器，将 MiniJava 通过五步变成可以在模拟器运行的 MIPS 汇编代码。这五步分别是：

- 类型检查：将输入的 MiniJava 代码通过建立符号表然后进行类型检查，对语法错误的代码报错。
- Piglet：此阶段是利用之前建立好的符号表，将 MiniJava 代码转成第一层中间代码表示。
- Spiglet：第二层中间代码表示，很像 Piglet 的中间代码。
- Kanga：开始有了寄存器分配，接近于目标代码的最后一层中间代码。
- MIPS：将 Kanga 代码转成最终的 MIPS 汇编代码。

通过自己开发一个编译器，可以加深理论课上的理解。虽然理论课和实习课是在同一学期上的，实习课的内容难免有些超前，但通过自行预习一些相关技术，能够更好的掌握理论课上所学的内容。在实现上，本课程采用的是 JavaCC 和 JTB 生成的语法树，而不是完全从空白做起，这在一定程度上减轻了我们的负担，不用去过多的关心底层的文法，使我们有更多精力放在重要的地方上。

1.2 环境配置

硬件环境：MacOS Sierra 10.12.1 64bit; 8GB RAM;
开发环境：Eclipse; 后期换成 IntelliJ IDEA 2016.3.1;
辅助工具：JDK 1.8; JavaCC & JTB; 各阶段模拟器等;

1.3 内容组织

接下来我会从每一阶段展开说明整个编译器的流程，会具体介绍本阶段目标、方法和难点等，也会在每个阶段的最后说明一下自身的不足之处。

2 类型检查

2.1 MiniJava 语法简介

MiniJava 是 Java 的一个子集，描述能力比 Java 弱。因为 Java 真实的运行过程是是从源代码编译成某个中间代码，然后在 Java 的虚拟机 (JVM) 中运行。所以为了方便起见，本课程采用了 MiniJava。MiniJava 的语法特征如下：（缺省遵从 Java 语法）

- 不允许方法重载
  - 一个类的定义中，不允许出现多个名字相同的方法名：不通过不同的参数来区分方法实现体，以简化编译器的实现
  - 但子类的方法自动覆盖父类定义的方法（多态）
  - 子类的属性也自动覆盖父类定义的属性
- 类中只能申明变量和方法（不能嵌套类）
- 只有类，没有接口，有继承关系（单继承）
- 不支持注释
- 一个文件中可以声明若干个类，即有且只有一个主类，辅类可以有多个，类不能声明为 public
- 表达式一共有 9 种：加、减、乘、与、小于、数组定位、数组长度、消息传递（即参数传递）、基本表达式
- 基本表达式一共有 9 种：整数、真、假、对象、this、初始化、数组初始化、非、括号表达式

2.2 Visitor 模式

初学 Visitor 模式是感觉很茫然、不理解，但经过第一次作业，会加深对 Visitor 模式的理解。看似麻烦不实用的开发模式，实际上会大大简化代码。

访问者模式的目的是封装一些施加于某种数据结构元素之上的操作。一旦这些操作需要修改的话，接受这个操作的数据结构则可以保持不变。访问者模式适用于数据结构相对未定的系统，它把数据结构和作用于结构上的操作之间的耦合分开，使得操作集合可以相对自由地演化。

实际上，我们用 JavaCC 生成了抽象语法树之后，Visitor 模式的框架就已经直接给我们写好了，我们只需要选择继承相关的类，然后在代码中填空就可以，非常方便。

2.3 符号表

2.3.1 符号表概述

第一次作业一个难点就是设计符号表，符号表不仅要在紧接着的类型检查中用，还要在翻译 Piglet 中间代码的时候用。一个好的符号表可以有效简化类型检查中的判断，符号表要尽量做到接口清晰、完善，结构清楚，否则将会在之后的开发中遇到很多麻烦以至于要不停的返工。

这里我的符号表的结构是这样的：

- AllType 类最高层的类，其中的方法有返回该 Type 的名字，所在行号和列号，这个类也是我后来在进行表达式整体传递时使用的。
- AllClasses 类，将代码中所有的类用 Hash 表记录下来，方法有插入和查找。
- MyClass 类，继承 AllType。对一个类的描述，记录父类名称、成员变量、成员方法等，以及之后要用到的方法有插入和查找成员变量以及成员方法，循环继承的判断等。
- MyMethod 类，继承 AllType。对一个方法的描述，记录返回值类型、所在类、局部变量，还有参数表等。
- BasicType 类，继承 AllType，记录基本类型，包括类作为的类型，判断类型是否兼容的方法。
- RealArgList 类，继承 AllType，顾名思义是在建立实参表的时候要用到的。

2.3.2 符号表细节

- MyClass 类

  MyClass 类主要是为了维护一个类中的所有信息，包括用来查询和插入成员变量和方法，在类型检查中要用的判断循环继承，查找父类，以及判断是否为一个类的子类等方法。

```
public class MyClass extends AllType{
    //此处的name指的是类的名字
    protected String parentClassName;
    public AllClasses allClasses;
    public Hashtable<String, BasicType> memberVar;
    public Hashtable<String, MyMethod> method;
    // 重复定义的类，那么除第一个类外都不能进行类型检查，否则会出现无意义的检查和报错
    protected boolean isRepeated;
```

```
    //构造函数
    public MyClass(String mClassName, AllClasses mAll, int rowNum, int colNum)
        {...}
    public boolean getRepeated() {...}
    public void setRepeated() {...}
    public String insertMemberVar(String varName, BasicType obj) {...}
    public BasicType getmemberVar(String varName) {...}
    public String insertMethod(String methodName, MyMethod obj) {...}

    //在本类中查找方法，只需给出类名字
    public MyMethod getMethod(String methodName) {...}

    //后期为了方便取消了继承重载，转Piglet只需要方法名在父类和本类中找方法
    public MyMethod getMethod_Piglet(String methodName) {...}

    //根据方法调用来寻找是否存在合适的方法，包括父类中的，根据方法名和参数类型表
    public MyMethod getMethod(String methodName, RealArgList methodArgu) {...}
    public String setParentName(String className) {...}
    public MyClass getParentClass() {...}
    public int checkLoop() {...}
    public boolean isSubClassOf(String mClass) {...}
    public BasicType findId(String curName) {...}
}
```

- MyMethod 类

  MyMethod 类主要用来维护一个方法中的所有信息，包括在哪个类中，参数表，局部变量，返回值类型等，在类型检查中需要检查参数表是否匹配以及返回值类型是否匹配。

```
public class MyMethod extends AllType{
    //此处的继承下来的name表示方法的名字
    protected BasicType methodType;
    public MyClass owner;
    public Hashtable<String, BasicType> localVar;
    protected Vector<BasicType> parameters;
    public Vector<String> paraName;

    // 在同一个类中重复定义的方法，那么除第一个方法外都不能进行类型检查，否则会出现无
        意义的检查和报错
    protected boolean isRepeated;
    public MyMethod(String mType, String mName, MyClass cOwner, int rowNum, int
        colNum) {...}
    public boolean getRepeated() {...}
```

```
    public void setRepeated() {...}
    public String insertLocalVar(String varName, BasicType varType) {...}
    public BasicType getLocalVar(String varName) {...}
    public void insertPara(BasicType varType) {...}
    public void insertParaName(String name) {...}
    public Vector<BasicType> getPara() {...}
    public BasicType getMethodType() {...}
    public boolean isEqualVector(RealArgList curArgList) {...}
    public boolean isEqualPara(MyMethod curMethod) {...}

    //判断是否存在重载
    public String override() {...}
    public BasicType findId(String curName) {...}
}
```

2.3.3 符号表的实现

我继承了带返回值和参数的 Visitor，来遍历语法树构造符号表，具体实现非常简单，只需要按照自己定义好的符号表的数据结构往里边添加内容就行了，这里不再赘述。

2.4 类型检查

2.4.1 检查的内容

通过语法分析只是代表程序满足语法要求，但是不一定满足语义要求，类型检查属于（浅）语义分析。类型检查的一些主要错误有：

- 使用未定义的类、变量、方法
- 重复定义了类、变量、方法
- 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
- 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
- 类的循环继承
- 不允许多继承（类 C 直接继承 B 和 A）
- 不允许方法重载（即一个类中定义若干同名的方法）

这里只列举了一些常见的错误。对于数组越界以及使用未初始化的变量和定义的变量未使用，在我的程序中是没有处理的，我认为这类错误只应该给出警告，而不应该算错误。为了方便，我就没有处理警告。事实上，C/C++ 编译器在默认情况下也不会给出类似的警告。

2.4.2 检查的时机

刚才说到构造符号表的时候需要遍历一次语法树，这次遍历过程中可以检查类、方法、变量的重复定义，多继承的检查，非常简单。

第二次遍历，对声明进行检查，主要查看是否有用没定义的类来声明的变量，本次遍历主要是为了防止之后的检查中对 ID 这个标识符混用的问题，因为一个类型可以被解释成标识符，一个变量的名字也可以被解释成标识符，所以为了代码的清晰，我单独做一次遍历来做声明类型的检查。

重点是第三次遍历，这里首先需要检查循环继承。先说一下循环继承如何检查，循环继承有好多种检查方式，有一种是将整个继承关系构造好之后进行拓扑排序，还有一种是对每个类去寻找父亲。这里我采用的是第二种方法，注意第二种方法有一个坑，就是可能存在 $A \to B \to C \to B$ 的情况，所以不能依据是否找回到了自己来判断存在循环继承，而是应该讲路径上的所以点做标记，然后判断是否遇到了已经被标记的点。

接下来就是方法中真实返回值和定义的返回值匹配的检查，这个过程也很简单。重点的检查都在后边对表达式的检查上，首先需要检查表达式左右两边的类型是否匹配，这部分的检查我放到了 BasicType 类的一个方法中，注意这里的匹配不一定是严格相等，有可能左边是右边类的父类。然后就是数组下标，if/while 语句判断和 Print 语句内部的类型检查。要说这部分的第二个难点，想必就是接下来过程调用的部分，过程调用一直都是这个编译器的难点，在此阶段，需要检查参数类型的匹配问题，以及需要找到被调用的方法在哪里，这些都需要很多的判断以及很多的细节，比如递归程序就是在 return 里加入过程调用，还有一种情况就是过程调用的结果当做一个参数，这些都需要仔细考虑，把这一部分解决了，基本的类型检查就算完成了。

2.4.3 调试技巧

这里简单说一下如何调试类型检查，首先一定要从小的程序开始测试，Factorial 的例子就非常好，包括了最基本的内容以及递归。然后可以去调试 BubbleSort 的带数组的部分，一般过了这两个大部分问题就没有了，接着可以去测试 TreeVisitor 之类的大 BOSS。

Java 这个语言对于异常处理是比较友善的，虽然可能大部分的错误是出现空指针 (NullPointer)，但他会将如何触发异常的过程打印出来，方便程序员去找到异常的位置，

相比于 C/C++ 来说会节省大量的调试时间。

2.5 类型检查小结

总的来说，第一部分还是很有难度的，毕竟是第一次接触 Visitor 模式，甚至有些同学可能还是第一次接触 Java 编程，自己摸索的过程也是充满着艰辛和痛苦。

简单说一下我的程序的亮点与不足，亮点是程序的鲁棒性很强，可以通过完整测试以及自己手动出的一些刁钻的例子，在打印错误提示中也尽量做到了和原 Java 报错相似，包括告知行号和列号以及出错语句等。不足之处在于，没有处理一些 Warning 的地方，例如数组越界，不初始化就使用和定义了未使用等警告。

3 Piglet

3.1 Piglet 简介

Piglet 是一种接近于中间代码的程序语言，它类似于三地址代码，但又不是严格的三地址，语法上支持复杂的表达式。它和最后的汇编很像，由一块一块组成，每一块都代表了一个过程。它与 MiniJava 的主要区别有：

- Piglet 取消了类的概念，取而代之的是一系列过程的组合。
- Piglet 开始出现内存的概念，有了地址空间的开辟和使用，不是简单的 new 操作。
- Piglet 取消了变量名，用 TEMP 加数字的方式来表示临时变量，注意块之间的 TEMP 可以重复，和寄存器还不太一样。
- Piglet 的传参也通过 TEMP 1 到 TEMP 20，最大只支持 20 个显式参数。

可以看到 Piglet 和 MiniJava 差距还是很大的，从代码形式上到逻辑上都和 Java 差距很大，这也给翻译带来了很大难度。

3.2 Piglet 翻译难点

3.2.1 类变量和方法

通过观察样例的 Factorial 代码，发现类成员变量调用通过一个一次间接指针得到实际地址，类成员方法通过一个二次间接指针得到地址。所以在遇到 new 一个新类的时候，先开辟一个 4 字节的空间给存放类成员方法数组的指针，然后紧接着存放一系列的成员变量。接下来再开辟一段地址空间存放类的成员方法。形式化的来说，假设 A 代表类的首地址，那么 $A \to B, x_1, x_2, x_3, \ldots$，其中 B 是指向方法数组的指针，x_i 是成员变量。$B \to y_1, y_2, y_3, \ldots$，其中 y_i 是成员方法。这样就可以用 A 这个临时变量来表示整个类了。

其中成员方法需要按照之前设计的符号表展开方法链，成员变量也是这样展开，展开的过程要注意重名方法的覆盖问题，距离近的优先。翻译成 Piglet 时，需要对方法进行重新命名，来消除重名的方法，我是使用“类名 + 下划线 + 方法名”作为新的方法名。

```
private void buildMethodList(MyClass curClass) {
    MyClass orgClass = curClass;
    Vector<String> curMethodList = new Vector<String>();
    Vector<MyClass> classList = new Vector<MyClass>();
    do {
```

```
        classList.add(curClass);
        curClass = curClass.getParentClass();
    }while(curClass != null);
    //先添加最祖先的方法，然后依次往后，为了之后的多态
    for(int i = classList.size() - 1; i >= 0; i--) {
        Enumeration e = classList.get(i).method.keys();
        while(e.hasMoreElements()) {
            String curMethodName = (String)e.nextElement();
            int x = contains(curMethodList, curMethodName);
            if(x == -1) {
                curMethodList.add(classList.get(i).getName() + "_" + curMethodName);
            }
            else {
                curMethodList.set(x, classList.get(i).getName() + "_" + curMethodName);
            }
        }
    }
    methodList.put(orgClass.getName(), curMethodList);
}
```

3.2.2 多态的处理

多态是面向对象的一个重要的机制，在 Java 中，多态是由 Java 虚拟机来实现的，而在本次实习中，我们不得不使用 C/C++ 使用的虚函数表来维护多态。这里的虚函数表也就是上段提到的方法链。对于如下代码：

```
A a = new B();
a.go();
```

这里的 B 是 A 的子类，两个类中都有 go 方法。a 所代表的对象其实是 B 类，所以调用的还是 B 中的 go 方法。

用之前设计好的方法链，new B 的过程中创建了 B 的地址空间，其中方法是按照上述方法排列好的，然后我们用 A 类来找 B 类的方法，正好对应的就是 B 中的 go。因为 B 方法链的前半截正好和 A 方法链相对应，通过这种方式就轻松实现了多态。

在做过程调用时，将代表类实体的临时标量作为第一个参数传递过去，这样就可以处理 this 的情况。

多态仅仅出现在 TreeVisitor 这个例子中，其他的例子没有出现多态。

3.2.3 数组的翻译

其实数组和之前类的翻译方法差不多，也是需要开辟一段地址空间。因为在 Minijava 的数组中要支持 length 方法，所以需要记录一下数组的长度。遇到 new 数组时，先开辟数组长度 +1 这么多的空间，然后将数组长度放到第一个位置就可以了。

3.2.4 参数的处理

Piglet 支持 20 个参数，除了第一个类实例临时变量的参数外，还有 19 个显式参数。所以如果参数个数小于等于 19 个，则直接传递。否则，需要空出最后一个参数的位置作为指向剩余参数数组的指针，类似于数组的处理就可以了。

3.3 Piglet 实现细节

这里我对 TEMP 编号是这样处理的，前 1-20 只用来存参数，参数不足 20 个也不再使用，接着从 21 开始往后编号。其中每个过程单独编号，这样不会影响，也能方便统一处理。

在每个方法开始的地方维护一个临时变量映射表，将方法中的变量映射到 TEMP 上，使用变量时到这个映射表中找。

接下来就是对着 Piglet 的语法描述去一步一步翻译了，翻译的过程虽然很繁琐，但总体来看还是比之前的类型检查轻松的，只需要将需要递归处理的地方递归处理即可。这里推荐用一个 Print 类来统一处理输出，这样格式输出比较美观，也方便排错。

注意：对 && 的翻译需要用 CJUMP 来处理，因为如果 && 前的表达式为假时，后边的表达式不会执行。

```
public AllType visit(AndExpression n, AllType argu) {
   //使用两次CJUMP实现跳转，避免副作用
   /* BEGIN
    *  MOVE TEMP t1 EXP1
    *  CJUMP TEMP t1 L1
    *  MOVE TEMP t1 EXP2
    *  CJUMP TEMP t1 L1
    *  MOVE TEMP t1 1
    *  JUMP L2
    *  L1 MOVE TEMP t1 0
    *  L2 NOOP
    *  RETURN TEMP t1
    * END
    */
   int t1 = curTmpNum++, L1 = curLableNum++, L2 = curLableNum++;
```

```
    PrintPiglet.pBegin();
    PrintPiglet.print("MOVE TEMP " + t1 + " ");
    n.f0.accept(this, argu);
    PrintPiglet.println("");
    PrintPiglet.println("CJUMP TEMP " + t1 + " L" + L1);
    PrintPiglet.print("MOVE TEMP " + t1 + " ");
    n.f2.accept(this, argu);
    PrintPiglet.println("");
    PrintPiglet.println("CJUMP TEMP " + t1 + " L" + L1);
    PrintPiglet.println("MOVE TEMP " + t1 + " 1");
    PrintPiglet.println("JUMP L" + L2);
    PrintPiglet.println("L" + L1 + " MOVE TEMP " + t1 + " 0");
    PrintPiglet.println("L" + L2 + " NOOP");
    PrintPiglet.pReturn();
    PrintPiglet.println("TEMP " + t1);
    PrintPiglet.pEnd();
    return null;
}
```

4 Spiglet

4.1 Spiglet 简介

Spiglet 是一种与 PigletPiglet 十分接近的中间代码。它主要特点是去除了嵌套表达式，使其十分接近的中间代码，与三地址代码更为接近，唯一的不同是可以使用无限的寄存器。我们主要的任务就是将嵌套多层的复合语句，剥离加入中转的 TEMP，翻译成一句句的简单语句。

总的来说，这次的作业最为简单，也算是期中季后的一次放松。

4.2 Spiglet 实现细节

我进行了两次遍历语法树，第一次很简单，只用来统计每个类用到的最大编号的 TEMP 是多少，然后记录下来给第二次遍历用。

第二次遍历就是一一对应的翻译了，翻译的重点也就是将复杂表达式展开，遇到复杂表达式时先递归进入然后处理成简单句后再返回，和英文中从句的处理很像。总体上就是递归处理即可。

这里可以尝试少产生冗余代码，例如已经是简单表达式了就没必要再展开一次。

```
public String visit(Exp n, String argu) {
    //防止产生冗余
    if(isMOVE) {
        isMOVE = false;
        return n.f0.accept(this, argu);
    }
    if(n.f0.which == 5 && allowInteger) {
        allowInteger = false;
        return n.f0.accept(this, argu);
    }
    allowInteger = false;
    if(n.f0.which == 4) return n.f0.accept(this, argu);

    //需要展开表达式
    int curNum = PrintSpiglet.TmpNum.get(argu);
    PrintSpiglet.TmpNum.put(argu, curNum + 1);
    PrintSpiglet.println("MOVE TEMP " + curNum + " " + n.f0.accept(this, argu));
    return "TEMP " + curNum + " ";
}
```

5 Kanga

5.1 Kanga 简介

Kanga 是一种与 SpigletSpiglet 十分接近的语言，它是一种面向 MIPS 的语言，与 Spiglet 的主要区别在于：

- 标号是全局的，而非局部的（对我来说不影响，我在 Piglet 里的标号已经是全局的）。
- 寄存器变成了有限的 24 个，基本和 MIPS 一一对应。
  - a_{0-3}：传参。
  - v_0, v_1：v_0 用来传递过程返回值，v_0 和 v_1 都可以用来作为表达式求值，以及栈中加载。
  - s_{0-7}：被调用者保存的寄存器。
  - t_{0-9}：调用者保存的寄存器。
- 开始有运行栈的管理。
- Call 指令的格式发生了较大变化。
- 每个过程的头部有三个参数，分别是参数个数、过程中需要栈单元的个数、过程中 Call 语句的最大参数个数。

5.2 Kanga 翻译难点

这次的难点在于如何将 Spiglet 中的 TEMP，一一对应成寄存器，需要用到寄存器分配算法，之后的纯翻译过程就比较容易了，只需要计算好过程头部的参数以及修改下 Call 语句就好。

5.2.1 寄存器分配

寄存器分配算法有很多，主要流行的是图染色和线性扫描。图染色效果很好，但耗时较高，代码难以调试。而线性扫描复杂度非常低，而且效果也不差，写起来也比较方便，所以这里我采用了线性扫描的寄存器分配算法。

第一步是基本块划分，这里我偷了一个懒，即每条语句都是一个基本块，这样便不用进行基本块的划分了。

接着就是建立流图，其中 CJUMP 和 JUMP 语句比较特殊，CJUMP 语句会有两个出边，一条是下一条语句，一条是跳向的语句。其余语句基本上就是直接指向下一句就可以了。

然后是进行活性分析，即分析每个变量的活性，对每条语句求 IN 和 OUT 集合即可，其中计算 IN 和 OUT 需要用到两个公式，即

$IN[B] = RIGHT[B] \cup (OUT[B] - LEFT[B])$

$OUT[B] = \cup_{n \in NEXT[B]}\ IN[n]$

其中 RIGHT 和 LEFT 分别是表达式的右值和左值集合，NEXT 是后继结点的集合，注意需要根据这个公式不断迭代直到不动点位置。

有了 IN 和 OUT 集合，就可以方便的进行活性区间分析了。为了方便我也做了保守估计，即每个变量的起始点是包含它的 IN 集合最小标号的语句，同理终止点是包含它的 IN 集合最大标号的语句，这样起始点和终止点的区间定位活性区间便不会出问题。其实线性扫描最好的办法是先做 SSA，但我为了方便也就没有做，留下了遗憾。

一切准备就绪后，便可以进行线性扫描的寄存器分配。大概思路很容易，就是先将所有区间按照起始点排序，然后依次分配寄存器，如果寄存器不够，就选择一个结束点最晚的溢出即可。下边是伪代码描述：

```
LinearScanRegisterAllocation
    active ← {}
    foreach live interval i, in order of increasing start point
        ExpireOldIntervals(i)
        if length(active) = R then
            SpillAtInterval(i)
        else
            register[i] ← a register removed from pool of free registers
            add i to active, sorted by increasing end point

ExpireOldIntervals(i)
    foreach interval j in active, in order of increasing end point
        if endpoint[j] ≥ startpoint[i] then
            return
        remove j from active
        add register[j] to pool of free registers

SpillAtInterval(i)
    spill ← last interval in active
    if endpoint[spill] > endpoint[i] then
        register[i] ← register[spill]
        location[spill] ← new stack location
        remove spill from active
        add i to active, sorted by increasing end point
    else
        location[i] ← new stack location
```

这里我还额外做了一些判断，如果一个变量的区间跨越了 Call 语句，那么优先分配 s 寄存器，因为 s 寄存器是被调用者保存。如果分配了 t 寄存器，那么每次 Call 之前还必须要保存，这样我觉得比较浪费。没有跨越 Call 语句的就优先分配 t 寄存器，t 寄存器不够再分配 s 寄存器。

5.3 Kanga 实现细节

这里我遍历了三次语法树，第一次是建立基本块，由于每条语句都是一个基本块，所以比较容易实现，过程中对每个块求 LEFT 和 RIGHT 集合，统计每个过程的语句个数并记录 Call 语句的位置，以及记录每个过程的参数个数。

第二次是建立数据流图，对于普通语句，连接当前与下一条边，对于 CJUMP 语句，连接两条出边，对于 JUMP 语句，连接跳转边。还对变量建立初始活性区间，按照之前 Piglet 的约定，临时变量号小于 20 的为形参，初始活跃区间的起点为 0；大于等于 20 的为普通临时变量，初始活跃区间的起点为当前语句 +1。

然后是之前所说的进行寄存器分配的过程。分配完成后进行最后一次遍历的翻译过程，此时翻译就变得很容易。需要注意，当传递的参数大于 4 时，需要用 PASSARG 来传递参数，参数标号从 1 开始。

下边是一个翻译的例子：

```
public String visit(HAllocate n, String argu) {
    String _ret = "HALLOCATE ";
    ProcInfo curProc = AllProc.get(argu);
    String argName = n.f1.accept(this, argu);
    if(argName.length() > 4 && argName.substring(0, 4).equals("TEMP")) {
        Integer curTempNum = Integer.parseInt(argName.substring(5, argName.length()));
        _ret = _ret + findTemp("v1", curTempNum, curProc, "USE");
    }
    else _ret = _ret + argName;
    return _ret;
}
```

5.4 Kanga 小结

到 Kanga 的翻译是本次实习的第二个难点，很能考察代码能力。本阶段令我满意的地方是我生成的代码都比较简短，去除了许多无用的代码。而遗憾和不足之处在于没有采用图染色算法写一遍，而且对于基本块的划分已经活性区间的分析过于简略。

6 MIPS

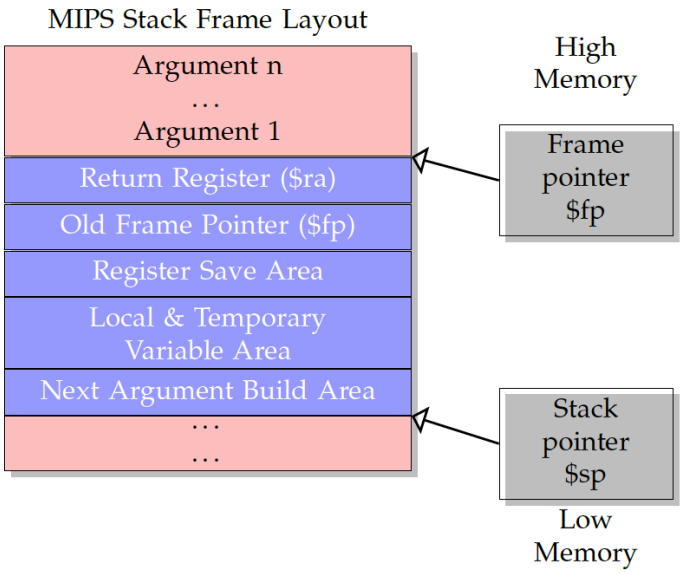
6.1 MIPS 简介

MIPS 是一种基于 RISC 的可实际运行的汇编代码，其重点语法特征已经指令格式已经在计算机组成中有所涉及，故这里不再详细展开说。这一阶段的翻译工作仅仅需要很少一部分的 MIPS 指令，因此即使不了解 MIPS 也能在较短时间内快速上手。在这里，调试和运行用的是 SPIM 模拟器。

MIPS 和 Kanga 最大的不同还是在于运行栈的设计上。回忆一下，Kanga 在每个过程头有三个参数，而这三个参数经过组合计算就可以对应 MIPS 每个过程开头需要申请多大的栈空间，我会在后边详细说明。设计好了运行栈，翻译就可以一一对应了，因为 Kanga 中的寄存器都是和 MIPS 一一对应的，MIPS 仅仅多出来了 $fp，$sp 和 $ra 操作栈帧和记录返回地址的寄存器。

6.2 MIPS 翻译难点

这一阶段唯一的难点就是在运行栈的设计和维护上，这里是我的运行栈的构造：



如图，对于一个正在运行中的过程，这个过程的栈帧可以分成五部分，第一部分是过程的返回地址 $ra，也就是过程结束后应当跳转到的位置；第二部分是保存旧的 $fp，这也是为了过程结束后能正确的恢复原来的栈帧；第三部分是保存本过程需要用到的 s 寄存器；第四部分是保存溢出的变量和 t 寄存器，由于我的 Kanga 寄存器分配不会把 t 寄存器分配给跨越函数调用的变量，所以就不需要保存 t 寄存器了；最后一部分是如果

本过程中出现了 Call，并且要 Call 的过程的参数超过了 4 个，那么就需要溢出参数到栈中，所以第五部分是准备溢出的参数。

可以用 Kanga 每个过程的头部三个参数来计算 MIPS 中每个阶段需要开大的运行栈，计算公式是：$a_2 - max(a_1 - 4, 0) + max(a_3 - 4, 0) + 2$，其中 a_i 是第 i 个参数，然后把得到的结果乘 4 就可以了。这样做的理由很简单，由于 Kanga 中的栈是包括本过程的溢出的参数，而不包括过程中 Call 的溢出的参数，而 MIPS 是需要为下一次 Call 做溢出参数的准备，所以将 Kanga 中的第二个参数，减去本阶段溢出的参数个数，然后再加上最大的 Call 溢出的参数的个数，最后给旧的 $fp 和 $ra 留出位置即可。

需要注意的一点是，为 Call 准备溢出参数的时候，需要以 $sp 为参考点，自下而上的存放参数。翻译 SPILLARG 的时候注意如何正确在栈中找到对应的位置就行了。

6.3 MIPS 实现细节

MIPS 指令集一定要弄清楚，例如 addi 和 addu，以及 sub 指令等，哪些可以带立即数，哪些不可以。从栈中取数可以用 $v0 和 $v1 来中转，这也和 Kanga 相对应。总之，这一步的实现有了之前的栈结构就很容易做了。

在翻译 MOVE 的时候，要根据后边运算的不同来选择不同的 MIPS 指令，稍微注意一下就好。

```
public String visit(HAllocate n, String argu) {
    String curExp = n.f1.accept(this, null);
    if(isNumeric(curExp)) PrintMIPS.println("li $a0, " + curExp);
    else PrintMIPS.println("move $a0, $" + curExp);
    PrintMIPS.println("jal _halloc");
    PrintMIPS.println("move $" + argu + ", $v0");
    return null;
}
```

最后整体上为了支持 SPIM 模拟器的运行，按照样例的格式将每个过程的.text 和.global 头写好，以及再手动写上 print 以及 halloc 函数等，直接抄样例就可以了。

7 阶段整合与总结建议

7.1 五合一

好了，到了现在每一阶段各自的部分就都完成了，但一个完整的编译器不可能要手动运行 5 次，所以最后一个任务就是将这个五个阶段整合在一起。我这里用的是 IO 重定向，因为之前都是用 Java 的 System.out.print 来输出的，所以直接将阶段输出重定向到临时文件，然后将输入也重定向到临时文件就可以了，用的是 System.setOut 和 System.setIn，然后整体的读入依然是标准输入，最后的输出也是标准输出。这样一个完整的编译器就完成了。

7.2 总结和建议

编译实习这门课极大的提升了我的代码能力，以及 Java 语言的编程能力。这也是我第一次接触和管理大型工程，除了第三次和第五次作业，其余每一次作业都要花上近 20 个小时来 code 和 debug，每一次也都会和助教讨论这一次作业中遇到了困难和困惑，我也有幸在课上进行了 4 次展示。

对于编译本身而言，书本上的知识永远只是理论，而实习课恰恰要动手将一个理论上的模型用代码来实现出来。当最后一个阶段完成，一个原生的 Java 代码到最后的 MIPS 汇编显示在屏幕上，整个人都很兴奋，一学期以来的疲惫一扫而空。以后我也还会去上其他实习课，而编译实习也给后续的实习课打好了一个基础，让我不再对大型的工程感到陌生和无助。

说起来遗憾的地方，也有很多，之前的小结中也提到过。由于理论课和实习课在一学期上，所以实习课的内容总是比理论课超前 2 周，导致我有的地方在似懂非懂的情况下就展开了编码，产生了许多困惑，代码效果也不是非常优秀，最后在理论课上才恍然大悟，但由于时间有限，我也不可能重新再按照正确的思路去重写一遍，这是很遗憾的。

这学期上课的教授和助教都非常好，助教帮助了我许多，老师在课上也是不断的激励我们去完成作业，但希望在课上能增加一些更多的演示，在第一次作业时，许多同学都不知道该如何生成需要的文件，不知道在哪里写代码，这些都极大的耽误了我们宝贵的时间，希望这些内容在以后的实习课中能讲的更清楚一些。

最后，感谢助教和老师一学期的辛勤付出，毕竟一学期不到 4 个月的时间就要从零开始完成一个完整的编译器，每个坚持到最后的人都是很棒的。也感谢所有在课上互动的同学，没有课上的交流答疑，我也不可能发现所有的 bug。写到这整个报告也就结束了，最后祝所有人期末顺利，新年快乐!