《编译实习》实习报告

史杨勍惟

1200012741 信息科学技术学院

2014-2015 学年第一学期

目录

综述 **3**

实习项目简介 3
已提供模板 3
功能与特点 4
设计流程简介 4

Step 1 类型检查 **6**

1.1 设计 7
1.2 实现 7
1.2.1 MyClasses 7
1.2.2 MyClass 8
1.2.3 MyMethod 9
1.2.4 MyBasicType 11
1.2.5 Visitor 设计 11
1.2.6 不足之处 13
1.3 测试 13

Step 2 miniJava 到 Piglet **14**

2.1 设计 15
2.2 实现 15
2.2.1 MethodTable 15
2.2.2 VarTable 16
2.2.3 继承关系处理 16
2.2.4 Visitor 设计 17
2.2.5 Distributor 18
2.3 测试 18

Step 3 Piglet 到 sPiglet **19**

Step 4 sPiglet 到 Kanga **20**

4.1 设计 21
4.2 实现 21
4.2.1 AllCFG 21
4.2.2 MethodCFG 21
4.2.3 CFGNode 22
4.2.4 Var 23
4.2.5 Visitor 设计 24
4.2.6 活性分析 24
4.2.7 线性扫描 25
4.3 测试 26

Step 5 Kanga 到 MIPS **27**

5.1 设计与实现 28
5.2 测试 28

Step 6 阶段整合 **30**

6.1 实现 31
6.2 测试 31

总结与建议 **32**

收获 32
建议 32

综述

实习项目简介

此次编译实习课程要求完成的是一个具有简单功能的 Java 编译器 miniJava。从整体来看，miniJava 完成的工作是以字符串形式读入一串 java 代码，然后检查代码是否有错误，如果没有错误则输出一个可直接在 MIPS 模拟器上执行的 MIPS 代码。从细节上看，miniJava 通过把整个翻译过程分成四个不同的阶段进行。在 java 代码和 MIPS 代码的中间，通过 Piglet，sPiglet 以及 Kanga 代码进行过渡。

已提供模板

在实现过程中，主要是在 UCLA 提供的一个基础模板上进一步实现的。此模板已通过 JavaCC 实现了各中间代码的语法分析和词法分析。我们需要做的是通过重写其中的一些 Vitisor 对相应的抽象语法树进行遍历，并在遍历的同时进行相应的数据结构的构建，错误的检查，以及翻译的工作。Visitor 和 Accept 是一个很有用的思想，这个思想的本质是:

将代码划分为对象结构和一个 Visitor 每个类中包含一个 accept 方法每个 accept 方法将 Visitor 作为参数一个 Visitor 为每个类包含一个 visit 方法计算过程中，控制流在 Visitor 的 visit 方法与具体对象中的 method 方法之间不断跳转

在我看来这个以提供的模板和编程模型是非常重要的，虽然从技术角度来说这些部分并不是编译中的关键步骤，有了这样的工具能够使我们的实现大大的方便了起来。要凭空写出一个真正的编译器是比较困难的，从某种角度上来说这次写出 miniJava 编译器实际算是一个“站在巨人的肩膀上”完成的编译器，因为已经有了很强的工具为我们节约了很多时间和精力了。

功能与特点

这次我完成的编译器有如下的特点：

- 能够检查绝大多数的语法错误，包括了：
  - 使用未定义的类，方法和变量
  - 重复定义的类方法和变量
  - 布尔表达式中的变量不为布尔类型
  - 运算表达式中的变量不为整型
  - 方法参数类型不匹配
  - 类的循环继承
  - 类的方法重载
  - **数组越界**
  - **使用未初始化的变量**
- 为对象实例，数组有着特殊的内存分配的安排
- 能够完善的支持 miniJava 所有支持的范式，并且对父子结构关系的处理非常完善。
- 进行了一些基本的优化，如短路代码，以及活跃分析和线性扫描使得寄存器分配的效果尽可能好。
- 代码输出使用了 On The Fly 的方式，节省了时间。

设计流程简介

此次编译器的实现步骤我完全参照了要求的五阶段进行，五个阶段的大致完成的功能如下：

- 第一步，类型检查，一共遍历了四次语法树，第一次为建立符号表，在符号表建立完毕后，更新符号表中的继承关系，然后顺便检查 A。第二次遍历检查是否有未定义的变量，第三次遍历检查是否存在类型错误，第四次遍历检查是否存在未定义的错误。
- 第二步，miniJava 到 Piglet 代码的翻译，这也是编译器从一个相对高级的语言进入一个比较机器化的语言的第一步，主要进行了变量的规范化以及循环的规范化。
- 第三步，Piglet 到 sPiglet，主要进行的是代码的进一步机器化，将一些复合的语句翻译为一些单一的语句。

- 第四步，sPiglet 到 Kanga，寄存器数量变得有限，并且加入了栈，这一步最主要的内容就是针对寄存器的分配方式。
- 第五步，Kanga 到 MIPS，MIPS 是直接运行在硬件上的代码，所以需要对一些细节进行进一步的实现，主要在于过程调用以及系统调用。

Step 1

类型检查

简介

Minijava 顾名思义是一个精简的 Java 语言，miniJava 没有复杂的复合表达式，只有一些简单的复合，new 语句和 call 调用的复合。另外，其支持的操作符也有限。

在继承关系方面，miniJava 最大的特点就是不支持函数的重载，即在任何的对象中都无法存在两个同名称不同参数类型的函数。不过 miniJava 支持重写，即父子类型可以存在同名不同参数类型的方法。

当然，虽然说和 Java 相比减少了很多，但是从编译的角度来看，整个实现的思路和框架和 Java 基本类似，所以说 miniJava 编译器的实现对我们理解编译会有很深的帮助。

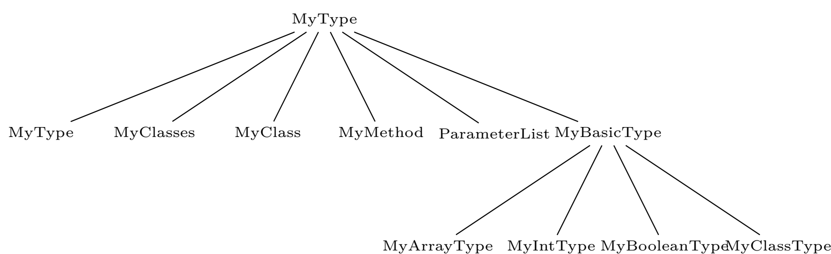
1.1 设计

总的来说检查的错误类型可以分为三类，一个是符号相关的错误，如重复定义，一个是和继承结构相关的，如重载错误，另外一个适合类型相关的，如参数类型不匹配或者布尔表达式的参数不是布尔类型等等。

- 第一次扫描是建立符号表，建立完符号表并且建立完继承关系的结构图后就可以对第二种情况进行检查，检查出循环继承以及函数重载等相关错误。
- 第二次扫描是检查未定义的错误类型，如果过程中出现了符号表中未定义的符号（包括过程中定义的，所属类成员，以及所属类成员的父类），那就存在未定义错误。
- 第三次扫描是检查类型是否存在错误，这个主要是通过类型结构在抽象语法树的传播来实现，并在某些特定的需要检查类型的节点上进行检查即可。
- 第四次是未初始化变量的检查。

1.2 实现

在符号表的构件中，我一共用了 10 个数据结构，包括最外层的父类结构 MyType，之后的 MyClasses 表示所有类，MyClass 表示一个类，MyMethod 表示方法，ParameterList 表示参数表类型，MyBasicType 是所有类型的抽象，MyIntType，MyArrayType，MyBooleanType，MyClassType 都是类型数据结构，具体关系图如下：



以下是几个重要类型的内部结构：

1.2.1 MyClasses

```
public class MyClasses extends MyType {
        protected Hashtable<String, MyClass> classes;
        //表示代码中包含的所有的类
        protected Hashtable<String, String> relations;
        //表示各个类之间的继承关系

        public boolean insertClass(String className, MyClass obj){...}
```

```
8          //插入类
9          public MyClass getMyClass(String className){...}
10         //获取类
11         public boolean checkLoop(){...}
12         //查询是否存在循环继承错误
13         public void checkType(){...}
14         //检查类型错误
15         public boolean checkOverride({...}
16         //查询是否存在方法重载错误
17         public void update(){...}
18         //更新类的继承关系以及错误查询
19         public boolean transClasses(MyClass obj) {...}
20         //让能访问到作用域的内容MyClassMyClasses
21 }
```

以上是所有类的数据结构，主要用来维护类间关系。

1.2.2 MyClass

```
1  public class MyClass extends MyType {
2          protected String className;
3          // 名字
4          protected Hashtable<String, MyClass> classes;
5          //保留上级的信息MyClasses
6          protected String parentName;
7          //父类名
8          protected Hashtable<String, MyBasicType> fields;
9          //所有成员
10         protected Hashtable<String, MyMethod> methods;
11         //所有方法
12         protected MethodTable methodTable;
13         //方法表（翻译时用到）
14         protected VarTable fieldTable;
15         //成员表（翻译时用到）
16
17         public boolean insertMethod(String methodName, MyMethod method){...}
18         //插入方法
19         public boolean transField(MyMethod method){...}
20         //把类内部成员传递给方法
21         public MyMethod getMethod(String methodName){...}
22         //获得方法名
23         public boolean insertField(String fieldName, MyBasicType fieldType) {...}
24         //插入一个成员
25         public MyBasicType getField(String fieldName){...}
26         //获得一个成员
27         public boolean hasField(String fieldName){...}
28         //是否包含成员
29         public String getParent(){...}
30         //得到父类名
31         public boolean setParentName(String parentName){...}
```

```
32          //设置父类名
33          public boolean checkType(Hashtable<String, MyClass> classes){...}
34          //检查类型是否存在
35          public boolean hasMethod(String methodName){...}
36          //是否包含方法
37          public boolean checkOverride(MyClass obj){...}
38          //检查重在错误
39          public boolean hasClass(String className){...}
40          //是否包含此类类型
41          public boolean addClass(String className, MyClass obj){...}
42          //加入此类类型
43          public MyClass getMyClass(String className){...}
44          //获得某类型
45          public boolean transClasses(){...}
46          //传递到每个方法MyClasses
47          public String getName(){...}
48          //获得此类名字
49          public boolean setName(String className) {...}
50          //设置此类名字
51          public MethodTable getMethodTable() {...}
52          //获取方法表
53          public VarTable getFieldTable(){...}
54          //获取成员表
55          public void fillTables() {...}
56          //填充成员表和方法表
57  }
```

这里的方法个数很多，主要原因在于遍历语法树过程中很多时候是在 MyClass 作用域内的，但是我们常常需要使用的 MyClasses 地方法，所以会把 MyClasses 的很多方法在 MyClass 中实现。

1.2.3 MyMethod

```
1   public class MyMethod
2           protected Hashtable<String, MyBasicType> vars;
3           //局部变量
4           protected ParameterList parameters;
5           //参数表
6           protected Hashtable<String, MyClass> classes;
7           //上上级的MyClasses
8           protected MyBasicType returnType;
9           //返回类型
10          protected String className;
11          //所属类
12          protected Vector<String> varInit;
13          //已初始化变量表
14          protected VarTable varTable;
15          //局部变量表
16
```

```
17          public boolean insertVar(String varName, MyBasicType varType){...}
18          //插入变量
19          public MyBasicType getVar(String varName){...}
20          //获取变量
21          public ParameterList getParas(){...}
22          //获取参数表
23          public boolean setParas(ParameterList paras){...}
24          //设置参数表
25          public boolean addPara(MyBasicType paraType){...}
26          //参数表中插入一类型
27          public MyBasicType getRet() {...}
28          //获得返回值
29          public boolean setRet(MyBasicType retType){...}
30          //设置返回值
31          public boolean checkType(Hashtable<String, MyClass> classes){...}
32          //检查类型是否存在
33          public boolean checkOverride(MyMethod method){...}
34          //检查重载错误
35          public boolean hasClass(String className){...}
36          //是否存在此类类型
37          public boolean addClass(String className, MyClass obj){...}
38          //添加一个类类型
39          public MyClass getMyClass(String className){...}
40          //获取一个类类型
41          public boolean hasVar(String varName){...}
42          //是否存在变量（包括类的成员和父类的成员）
43          public boolean hasLocal(String varName){...}
44          //是否存在局部变量
45          public boolean subClass(MyBasicType t1, MyBasicType t2){...}
46          //类型是否存在继承关系
47          public boolean typeMatch(MyBasicType t1, MyBasicType t2){...}
48          //类型是否可以互相匹配
49          public String getName(){...}
50          //获取名字
51          public boolean setName(String className){...}
52          //设置名字
53          public boolean isInit(String varName){...}
54          //查看变量是否初始化
55          public void insertInit(String varName){...}
56          //插入已初始化变量
57          public int getParaNum(){...}
58          //获取参数个数
59          public VarTable getVarTable() {...}
60          //获取变量表
61          public boolean matchType(ParameterList paras){...}
62          //此过程的参数列表是否与另一参数列表匹配
63  }
```

和 MyClass 一样，MyMethod 中也实现了很多本应该在 MyClasses 中实现的方法，原因也一样，很多时候的检查都是在 MyMethod 作用域而不是 MyClasses 作用域

1.2.4 MyBasicType

```
public class MyBasicType {
        protected String typeName;

        boolean sameType(MyBasicType type){...}
        //两类型是否相同
        public String getType(){...}
        //获取类型名
        public boolean isBasic(){...}
        //是否基本类型
}
```

在此基础上，MyClassType，MyIntType，MyBooleanType，MyArrayType 本质上就是对 typeName 的一个封装，但是将它们抽象成数据结构可以更好的进行抽象语法树上的类型传递，通过 instanceOf 就能判断究竟是什么类型。

1.2.5 Visitor 设计

总共对抽象语法树进行了四次遍历：

- 第一次遍历（构建符号表）：

  构建符号表的起点是一个空的 Classes 类，每次遇到定义 class 语句就把 class 相关的信息加入 Classes，随后切换作用于到 Class，在 Class 内部每次访问到定义成员的节点就将符号插入到成员列表中，每次访问到过程定义就将过程插入到 class 中，再把作用域切入过程。切入过程后首先需要把形参的符号加入符号表，其次再把所有访问到过程中变量定义的符号插入到过程的符号表中。在这个过程中就可以完成函数重载以及变量名重定义的检查了以及同类之间的重载错误。

- 更新符号表（第一次检查）：

  在第一次构建语法树的遍历中的过程中会更新类之间的继承关系，主要通过访问 classextension 的节点时候构建。但是此处只是简单的把父类插入了子类。所以在 update 的过程中需要构建完整的继承关系图。构建完后就可以检查是否有方法重载了是否有循环继承以及类不存在错误和重载错误这里的重载是指子类和父类中的方法出现同名的情形但是没有类型参数不匹配的错误，和之前的重载错误检查有一定的不同之处。

- 第二次遍历（检查未定义错误）：

  这里检查未定义错误，主要检查是否调用了未定义的方法以及在非定义的语句中使用了未定义的符号。只要调用检查的接口即可。

- 第三次遍历（检查类型错误）：

  这里检查类型错误，主要检查是否存在类型错误。在访问语句节点的同时，visitor 的 accept 函数会返回一个 MyBasicType 类型，这个类型表示了每个变量的类型，当我们在一些语句中访问到错误类型的时候，如布尔表达式的变量不为布尔类型或者运算表达式的变量不为整型，以及调用过程的参数不匹配等等，就会报错。

- 第四次遍历（检查变量未初始化错误）：

  这是我自己新添加的一个检查，即检查是否会在某语句中使用未初始化的变量。由于类中的成员是否初始化是无法静态检查的，因为很有可能在其他方法中被初始化，所以这里我仅仅检查了方法中的临时变量。检查的方法是访问 Identifier 节点时判断这个 Identifier 是否被初始化了，在访问 Assignment 以及 Allocation 的节点处把相应的 Identifier 初始化，这里要注意的一点是对于 Assignment 的访问要先访问右边再访问左边，因为未初始化的错误往往出现在右边，如果先访问左边就可能已经将变量初始化了，导致无法检查出错误。

下表是对于各种类型错误的检查点，包括遍历点和抽象语法树的节点：

错误入口		
错误描述	**阶段**	**语法树节点**
类型重复定义	BuildSTVisitor	ClassDeclaration
过程重复定义	BuildSTVisitor	MethodDeclaration
变量重复定义	BuildSTVisitor	VarDeclaration
成员重复定义	BuildSTVisitor	VarDeclaration
循环继承	MyClasses.checkLoop()	/
方法重载	MyClasses.checkOverride	/
变量未定义	CheckUndefinedVisitor	AssignmentStatement
变量未定义	CheckUndefinedVisitor	PrimaryExpression
数组变量未定义	CheckUndefinedVisitor	ArrayAssignmentStatement
调用过程未定义	CheckUndefinedVisitor	MessageSend
类型未定义	CheckUndefinedVisitor	AllocationExpression
左右值类型不匹配	CheckTypeErrorVisitor	AssignmentStatement
数组变量下标类型不匹配	CheckTypeErrorVisitor	ArrayAssignmentStatement
数组长度或索引类型不匹配	CheckTypeErrorVisitor	Array(Length/Lookip)
布尔表达式类型不匹配	CheckTypeErrorVisitor	(While/If)Statement
布尔表达式类型不匹配	CheckTypeErrorVisitor	(And/Not)Expression
运算表达式类型不匹配	CheckTypeErrorVisitor	(Plus/Misnus/Times)Expression
比较表达式类型不匹配	CheckTypeErrorVisitor	CompareExpression
过程参数或返回值不匹配	CheckTypeErrorVisitor	MessageSend
变量未初始化	CheckInitializationVisitor	Identifier

这里我还维护了一个 PrintError 的数据结构，主要用来收集所有的出错信息并在最后一起输出。

1.2.6 不足之处

一开始我并未考虑效率问题，所以设计了很多次遍历已完成不同的功能，但是现在发现这样的效果不太好。其实后面两次遍历完全可以整合成一次，而第二次和第三次整合到一起有些麻烦，主要是当类中找不到方法的时候作用域不能往下传递，但是这个要整合其实也是可以的，只要稍微加一些判断就行了。我觉得这里的设计不足也是我整个编译器设计中最大的一个缺陷了，以后会尽量控制语法树的遍历次数，设计出更高效的语法检查程序。

1.3 测试

类型检查的测试我主要使用的是老师给的那 100 个测试数据，因为那 100 个测试数据基本涵盖了大部分的类型错误，在调试过程中我发现了我大部分的问题都出在过程中寻找变量是否存在的过程中。当过程不存在本地变量时需要从过程所属的类的成员中寻找，再找不到还需要到父类的成员中寻找。这里有两种方法，一种是更新的时候把父类的成员名复制给子类，另一种是寻找的时候在子类中找不到后在到父类中寻找，一个主要时间在更新中，另一个主要时间开销在检查中。我选择的是后者。其中没有出现的应该只有循环定义，我构造了一个循环定义的例子，也很顺利地通过了。不过后来我发现这一步我忽视了一点，就是当方法调用时比较参数列表时，如果两个参数对应类型存在继承关系，那么也应该视为符合条件的调用，这一点我一开始没有注意到，所以导致了有些合法程序会误报。这个也是我在最后整合的时候发现的，并且进行了修复。

Step 2

miniJava 到 Piglet

简介

这个是第一步翻译，在我看来也是改变最多的一次翻译，因为这是一次从高级语言到一个类机器语言的转变过程，这里的翻译需要将变量翻译成寄存器，并且要为高阶的数据类型设计内存结构，如方法，类型，数组等等。

与此同时，参数个数也有了限制，不能传递超过 20 个参数，使得要为 20 个参数设计特殊的翻译方法。

2.1 设计

这一步最主要的是内存结构的设计，我的内存结构设计如下：

- 对象实例的内存结构如下，

对象实例内存结构	
Offset	**内容**
0	方法表地址
4	第一个成员
8	第二个成员
...	...

- 方法表：

方法表内存结构	
Offset	**内容**
0	第一个方法
4	第二个方法
...	...

- 数组的构建方法如下

数组结构				
长度	a[0]	a[1]	...	a[n]

- 多余 19 个参数时保留前 18 个参数，TEMP19 保存一个地址，改地址为后面所有参数保留了空间。

2.2 实现

2.2.1 MethodTable

我的方法表内部情况如下：

```
1 public class MethodTable {
2         protected Hashtable<String, String> labels;
3         //方法名到标签名的映射
4         protected Vector<String> list;
5         //方法表
6
7         public Hashtable<String, String> getLabels(){...}
8         //获得方法标签表
9         public String getLabel(String name){...}
```

```
10          //根据方法名获得方法标签
11          public String getLabel(int index){...}
12          //根据偏移量获得方法标签
13          public int getOffset(String name){...}
14          //根据方法名获得偏移量
15          public void addMethod(String name, String labelName){...}
16          //加入方法
17          public int getMethodNumber(){...}
18          //获得方法个数
19  }
```

2.2.2 VarTable

我的变量表（包括方法的局部变量表和类型的成员表）的内部结构如下：

```
1   public class VarTable {
2           protected Vector<String> list;
3           //变量表
4           protected Hashtable<String, Integer> offsets;
5           //偏移量表
6
7           public Hashtable<String, Integer> getOffsets(){...}
8           //获得偏移量表
9           public int getOffset(String name) {...}
10          //获得变量偏移量
11          public int getVarNumber(){...}
12          //获取变量个数
13          public void addVar(String name){...}
14          //添加偏移量
15  }
```

2.2.3 继承关系处理

这里一个比较棘手的问题就是父子关系的处理，我们知道子类的实例既要为自己的成员和方法预留空间，还要为他所有的父类的成员和方法预留空间。由于不支持重载，所以子类中如果出现和父类同名的方法，可以直接覆盖掉原方法，但是如果子类出现与父类同名的成员依然需要保留父类的成员。

上述代码中，子类会首先调用父类的 x，然后在依次更新自己的方法和成员，对于方法，如果出现同名，直接在修改 label 名即可而不需要修改方法的 offset，对于成员，子类会在来的成员表后新添加这个成员，并且把这个成员的 offset 修改为表最后的位置。但是此时父类的同名成员依然在表中，只不过是所有的子类方法无法访问了。子类如果出现和父类同名的成员，访问该父类成员只能通过父类方法进行，在父类方法中这个成员的 offset 并不是列表的最后，而是原来的该成员的位置，我们可以看到子类的成员列表的前缀和父类的成

员列表完全一致，这样父类方法就可以在子类实例中访问到父类成员了，这个设计也是十分巧妙的地方。

2.2.4 Visitor 设计

Visitor 的主要设计思路是把 miniJava 的语句翻译成简单的三地址代码的语句，这里需要注意的地方如下所述：对于过程中每一个语句的左值变量有两种情况，一种是过程中定义的临时变量，这个变量只需要从过程的 VarTable 中找到对应的偏移量即可，另一种是方法所属类的成员，这样就要从方法的第 0 个参数，即 this 指针所对应的对象实例的内存结构中中找到相应 offset 位空间的数据即可。对于逻辑语句 A&&B 的翻译，我才用了短路代码的设计，具体代码如下：

```
BEGIN
MOVE value 1
CJUMP A Label
CJUMP B Label
MOVE value 0
 Label:
RETURN value
END
```

与此同时，关于数组是否越界的检查也是在此处进行的，我的 ArrayLookup 的节点的遍历代码如下：

```
public MyType visit(ArrayLookup n, MyType argu) {

            ...

      PrintBoard.println("CJUMP LT ");
      PrintBoard.println(indexTemp);
      PrintBoard.println("BEGIN");
      PrintBoard.println("HLOAD " + temp + " " + arrayTemp + " 0");
      PrintBoard.println("RETURN " + temp);
      PrintBoard.println("END");
      //检查长度是否超出
      PrintBoard.println(errorLabel);
      //超出跳入Error
      PrintBoard.println("JUMP " + correctLabel);
      PrintBoard.println(errorLabel + " ERROR");
      PrintBoard.println(correctLabel + " NOOP");
      //否则再获取内容
      PrintBoard.println("HLOAD " + temp + "PLUS TIMES 4 "
        + indexTemp + " " + arrayTemp + " 4");
      PrintBoard.println("RETURN " + temp);
      PrintBoard.println("END");

            ...

```

```
    }

}
```

2.2.5 Distributor

这里我还维护了一个 Distributor 数据结构作为分配器，用处是为每个过程分配临时寄存器 Distributor 代码如下：

```
public class Distributor {
        int tempCount;
        static int labelCount;

        public Distributor(MyMethod method) {
                tempCount = method.getVarTable().getVarNumber()+1;
        }
        public Distributor() {
                tempCount = 0;
        }
        public String newTemp() {
                //System.out.println("here");
                String ret = new String("TEMP " + tempCount);
                tempCount++;
                //System.out.println(ret);
                return ret;
        }
        public String newLabel() {
                String ret = new String("LABEL"+labelCount);
                ++labelCount;
                return ret;
        }
}
```

这个维护非常简单，只需要通过递增的方法为每个方法维护就行了。

2.3 测试

此步我使用的是 UCLA 项目主页上的那八个样例，这一步我的错误挺多，不过主要出现还是在内存结构的设计上的问题，比如偏移差了 4 位，以及子类为父类的成员和方法预留的空间等等。与此同时我还构造了多余 20 个参数以及恰好 20 或 19 的函数，同时还添加了数组越界的检查，以测试过程结构的正确性。

Step 3

Piglet 到 sPiglet

sPiglet 和 piglet 非常相似，最主要的不同点在于 sPiglet 不支持复合的表达式，支持 simpleEXP，所以需要做的就是在 piglet 的抽象语法树的遍历过程中吧复合语句的节点一个一个拆开就行。这里就不展开详述了。

实现过程中不需要专门设计其他的数据结构了，只需要用 String 简单地表示一下寄存器即可。对于每次访问节点 stmt 及其以下节点的时候将其翻译为简单语句在返回一个新的寄存器（以字符串形式）即可，这里需要维护一个分配寄存器的工具，原则上来说分配的起点应该是 Piglet 寄存器中最大的序号并且以递增的方式分配，我在这里偷了个懒，因为寄存器序号是 Integer literal 所以我直接以 0x7FFFFFFF（整型上界）作为起点以递减的方式分配了，这样可以方便一点。

Step 4

sPiglet 到 Kanga

简介

kanga 和 spiglet 最大的不同点在于 kanga 的寄存器是有约束的，他的寄存器的个数有限，并且是遵守 MIPS 规范的，不同的寄存器也有着不同的功能。所以此步翻译最大的特点就是需要进行寄存器分配以及栈的维护。与此同时，kanga 在方法头部需要约定该方法使用的寄存器个数以及栈的深度，这也是翻译是需要维护的内容。设计：寄存器分配我是用的是活性分析和线性扫描的方式，活性分析就是需要对每一个变量计算他的活跃区间，而线性扫描就是对找到每一个节点所有已着色且活跃的变量中结束时间最晚的那个，将其溢出，把其颜色分给新的变量。活性分析和线性扫描均是在控制流图上进行的，所以我们需要首先通过一次抽象语法树的遍历进行

4.1 设计

寄存器分配我是用的是活性分析和线性扫描的方式，活性分析就是需要对每一个变量计算他的活跃区间，而线性扫描就是对找到每一个节点所有已着色且活跃的变量中结束时间最晚的那个，将其溢出，把其颜色分给新的变量。活性分析和线性扫描均是在控制流图上进行的，所以我们需要首先通过一次抽象语法树的遍历进行

4.2 实现

首先我们需要设计一个能够表示控制流图的数据结构，在我的代码中，AllCFG 表示所有过程的控制流图，MethodCFG 表示单个过程的控制流图，CFGNode 表示控制流图中的一个节点，分别表示如下

4.2.1 AllCFG

```
1  public class AllCFG {
2          protected Hashtable<String, MethodCFG> methodGraphs;
3          //所有的方法控制流图
4  
5          public MethodCFG getMethodGraph(String name){...}
6          //获得一个方法控制流图
7          public void addMethodGraph(MethodCFG methodGraph){...}
8          //添加一个方法的控制流图
9  }
```

4.2.2 MethodCFG

```
1  public class MethodCFG extends MyType{
2          protected String name;
3          //方法名
4          protected int paraNum;
5          //变量个数
6          protected Vector<CFGNode> nodeList;
7          //所有的控制流节点
8          protected Hashtable<String, Integer> labelIndex;
9          //标签到控制流节点索引的映射
10         protected Hashtable<Integer, Var> varList;
11         //寄存器号到变量数据结构的映射
12         public static int MAX_COLOR = 8;
13         //最大色数
14         protected Vector<Boolean> colorList;
15         //累积的颜色使用情况
16         protected Vector<Boolean> usedList;
```

```
17          //当前的颜色使用情况
18          protected int maxCallParaNum;
19          //调用方法的最大参数个数
20          protected int spillNum;
21          //溢出的个数
22
23          getters & setter ...,
24          public void updateMaxCallParaNum(int paraNum){...}
25          //更新色数
26          public void addNode(CFGNode node)
27          //添加节点
28          public CFGNode getNode(String labelName)
29          //通过标签名获得节点
30          public CFGNode getNodeAt(int i)
31          //通过语句序号获得节点
32          public CFGNode getFirstNode()
33          //获取第一个节点
34          public CFGNode getLastNode()
35          //获取第二个节点
36          public int getUsedColorNum()
37          //获取使用的颜色数
38          public boolean checkColor(int color)
39          //检查颜色是否被使用
40          public Var getVar(int tempNum)
41          //获取变量
42          public void update()
43          //更新方法
44          private void connectAll()
45          //连结所有节点
46          public void livenessAnalysis()
47          //活性分析
48          public void linearScan()
49          //线性扫描
50  }
```

4.2.3 CFGNode

此处主要使用的是编译原理中所述的基于 DU 链的活性分析：

```
1   public class CFGNode extends MyType {
2           protected HashSet<Integer> def;
3           //DEF
4           protected HashSet<Integer> use;
5           //USE
6           protected HashSet<Integer> in;
7           //IN
8           protected HashSet<Integer> out;
9           //OUT
10          protected Vector<CFGNode> pre;
11          //前驱集合
```

```
12         protected Vector<CFGNode> suc;
13         //后继集合
14         protected String label;
15         //节点标签
16         protected String jumpLabel;
17         //目标标签（如果是跳转语句）
18         protected String jumpStyle;
19         //跳转类型（或）JUMPCJUMP
20         protected MethodCFG method;
21         //所属方法
22 
23         getters & setters
24         public void addPre(CFGNode node){...}
25         //添加一个前驱
26         public void addSuc(CFGNode node){...}
27         //添加一个后继
28         public void addUse(int num){...}
29         //添加一个使用量
30         public void addDef(int num){...}
31         //添加一个定义量
32         public void addLive(int num){...}
33         //添加一个活跃变量
34         public void removeLive(int num){...}
35         //删除一个活跃变量
36         public void update(){...}
37         //更新节点信息
38 }
```

4.2.4 Var

这里用到了辅助的数据结构 Var 记录类型信息

```
1  public class Var {
2          protected int tempNum;
3          //寄存器号
4          protected int begin;
5          //起始点
6          protected int end;
7          //中止点
8          protected int color;
9          //被染的颜色
10         protected int spillNum;
11         //溢出的偏移量
12 }
```

4.2.5 Visitor 设计

构造数据流图的起点是一个新的 AllCFG，在遍历语法树的过程中每次访问过程节点的时候就将作用于转换为一个新的 MethodCFG，并在过程节点退出的时候调用 update 函数对此过程进行活性分析和线性扫描。这里用到的另一个巧妙的方法在于对 NodeSequence 的遍历，纵观整个 sPiglet 的 BNF 范式，只有在对（Label?Stmt）遍历时才会用到 NodeSequence，这样我们就可以利用 NodeSequence 作为 MethodCFG 到 CFGNode 的作用域转换点了，以及不同 CFGNode 的切换点就可以了。在遍历到 JUMP 和 CJUMP 语句的时候我们分别为此时的 CFGNode 的后继结点进行特殊处理，对于 JUMP 只需找到 Label 处的 CFGNode 即可，对于 CJUMP 需要同时添加下一条语句和 Label 处语句。至于其他的语句，只需要简单的加入下一条语句即可

4.2.6 活性分析

主要根据编译原理龙书上的活性分析的公式进行设计：

$$IN[EXIT] = \emptyset \tag{4.1}$$

$$IN[B] = use_B \cap (OUT[B] - def_B) \tag{4.2}$$

$$OUT[B] = \cap_{S\ B} \quad IN[S] \tag{4.3}$$

以下是活性分析的核心代码：

```
1
2  CFGNode node = nodeList.elementAt(i);
3  HashSet<Integer> origin = new HashSet<Integer>(node.getIn());
4  for(CFGNode sucNode : node.getSuc()) {
5          for(int tempNum : sucNode.getIn()) {
6                  varList.put(tempNum, new Var(tempNum));
7                  node.addLive(tempNum);
8          }
9  }
10 for(int tempNum : node.getDef()) {
11         node.removeLive(tempNum);
12 }
13 for(int tempNum : node.getUse()) {
14         node.addLive(tempNum);
15         varList.put(tempNum, new Var(tempNum));
16 }
17 if(!node.getIn().equals(origin)) {
18         flag = false;
19 }
```

4.2.7 线性扫描

线性扫描主要思想就是当颜色不够时溢出最远终止生命周期的变量

以下为核心代码：

```

for(int i = 0; i < nodeList.size(); ++i) {
        CFGNode node = nodeList.elementAt(i);
        for(int tempNum : node.getIn()) {
                Var var = varList.get(tempNum);
                liveList.add(var);
        }
        HashSet<Var> removeList = new HashSet<Var>();
        for(Var var : liveList) {
                if(var.getEnd() < i) {
                        int color = var.getColor();
                        if(!var.isSpilled()) colorList.setElementAt(false, color);
                        removeList.add(var);
                }
        }
        for(Var var : removeList) {
                liveList.remove(var);
        }
        //以上为移除已过期变量
        for(Var var : liveList) {
                if(!var.hasColor() && !var.isSpilled()) {
                        int availableColor = -1;
                        for(int j = 0; j < MAX_COLOR; ++j) {
                                if(colorList.elementAt(j) == false) {
                                        availableColor = j;
                                }
                        }
                        if(availableColor == -1) {
                                //无可用颜色
                                int maxEnd = -1;
                                Var spillVar = new Var(-1);
                                for(Var tempVar : liveList) {
                                        if(!tempVar.isSpilled()
                                         && tempVar.hasColor()
                                         && tempVar.getEnd() > maxEnd) {
                                                maxEnd = tempVar.getEnd();
                                                spillVar = tempVar;
                                        }
                                }
                                var.setColor(spillVar.getColor());
                                spillVar.setSpill(spillNum++);
                                //溢出变量
                        } else {
                            //有可用颜色
                                colorList.setElementAt(true, availableColor);
```

```
46                                  var.setColor(availableColor);
47                                  usedList.setElementAt(true, availableColor);
48                          }
49                  }
50          }
51  }
```

4.3 测试

我使用的依然是 UCLA 那 8 个程序，在测试过程中我发现的一个问题是关于临时寄存器的使用，我一开始用的都是 v0，后来发现有些语句有两个参数，不能都用 v0 作为寄存器。我采用的方法是使用了一个 turn 值，每次分配一个临时寄存器就反转 turn 值，并根据 turn 来决定返回哪个寄存器，这样就不会产生冲突了。

Step 5

Kanga 到 MIPS

简介

最后一步翻译就是最后通向机器语言的世界的一步了。MIPS 代码我在上学期的计算机组成课上掌握了其大致框架。作为 RISC 代码，MIPS 语言有着很强的规范性，也没有很复杂的语句。总的来说和 kanga 代码非常的相近。这里我们要做的就是维护 MIPS 的栈帧结构以及一些相应的细节实现，使得我们的代码能够真正的在机器上执行。

5.1 设计与实现

kanga 到 mips 的改变基本就是栈帧结构的维护，MIPS 的栈帧结构如下所示：

MIPS 栈帧结构	
Address	**内容**
-8($sp)	上帧地址
-4($sp)	返回地址
0($sp)	栈起始
...	...

在每次函数调用前和函数调用后需要将加入返回地址以及前帧地址的入栈出栈操作。另外由于是直接和硬件交互，所以 ALLOCATE 和 PRINT 就需要用相应的系统调用来完成了，具体实现的时候将 PRINT 和 ALLOCATE 分别作为两个独立的 MIPS 方法，在方法中进行系统调用即可。如下所示：这里也没有构造特别的数据结构，还是通过 String 作为寄存器得标识在抽象语法树上进行遍历，以表示每个节点返回的寄存器号。在访问抽象语法树的过程中比较重要的就是需要在访问方法的头部和尾部进行栈的维护，头部需要添加的是：

```
1
2  if(maxCallParaNum > 4) maxCallParaNum -= 4;
3  else maxCallParaNum = 0;
4  if(maxParaNum > 4) maxParaNum -= 4;
5  else maxParaNum = 0;
6  stackSize += (maxCallParaNum+2);
7  stackSize *= 4;
8
9  System.out.println("sw $fp, -8($sp)");
10 System.out.println("sw $ra, -4($sp)");
11 System.out.println("move $fp, $sp");
12 System.out.println("subu $sp, $sp, " + stackSize);
```

尾部需要添加的是：

```
1 System.out.println("addu $sp, $sp, " + stackSize);
2 System.out.println("lw $fp, -8($sp)");
3 System.out.println("lw $ra, -4($sp)");
4 System.out.println("j $ra");
```

5.2 测试

我使用的是 UCLA 的例子，这里出现了一个小问题，由于在这一步中我一开始用 v1 作为临时寄存器，这样会导致在如果 kanga 代码的过程中 v1 被使用，那么其值就会出错。所以这里需要找一个不曾再 kanga 中出现的寄存器作为临时寄存器使用。我一开始想到的

是 s7，由于前面的着色数是可以设置的，那么如果我减少前面的颜色的话可以腾出 s7 作为 kanga 到 MIPS 翻译过程中的临时寄存器。这样虽然整个编译器是没有问题的，但是在课程测试时，kanga to MIPS 作为独立的工具就会有问题，因为 kanga 程序中肯定会存在 s7。最后，我发现了 MIPS 还是有未利用的寄存器 $gp，用它作为临时寄存器既可以不改写前面着色数，有解决了冲突问题，是一个很不错的解决方案。

Step 6

阶段整合

简介

最后就是要将五个步骤整合到一起了，由于之前都是直接用 System.in 到 System.out 的操作，这个类似于编译原理课上所说的 On The Fly 方法，即不需要保存 code 属性，只需要在遍历抽象语法树的时候直接输出即可，但是整合到一起的时候不能直接输出，所以需要对 System.out 做一个数据结构的抽象，所以这一步需要对 System.out 进行保存。我设计了一个全局的 PrintBaord。

6.1 实现

PrintBoard 的结构如下：

```
public class PrintBoard {
        private static String code = "";
        public static void clear() {
                code = "";
        }
        public static void println(String s) {
                code += s;
                code += "\n";
        }
        public static void print(String s) {
                code += s;
        }
        public static String getCode() {
                return code;
        }
        public static void output() {
                System.out.print(code);
        }
}
```

可以看到，PrintBoard 只是对于 System.Out 的一个简单的包装而已，它把字符串保存在了类内部，而不是直接打印，以便后续使用。然后在仿照每个步骤的 main 方法改写出一个 translate 方法，这个方法从 Print 的字符串读取字符，并且清空这个字符串，然后把各个 Visitor 中的 System.out 都改成 PrintBoard（除了最后一步翻译），这样就可以用 PrintBoard 把五个步骤的输入输出很好的连接起来了（如下表）

输入输出控制		
阶段	**输入**	**输出**
Java->Piglet	System.in	PrintBoard
Piglet->sPiglet	PrintBoard	PrintBoard
sPiglet->Kanga	PrintBoard	PrintBoard
Kanga->MIPS	PrintBoard	System.out

6.2 测试

在第一次把五个阶段整合到一起的时候，我的编译器并未正常运行，问题竟然出在是类型检查上。其实我们之前测试都是用 UCLA 的样例测试的，而不是用自己上一阶段所跑出来的结果，所以有的时候一些问题会被隐藏掉。而五个阶段整合到一起的调试也不方便，所以在这个方面我还是花了一些时间的。

总结与建议

收获

- 对于编译器的设计有了一个更深层次的认识，与此同时也对编译原理知识有了进一步的巩固，说实话开学的时候上学期编译的知识已经有点忘记了。
- 发现了代码实现是非常困难的。其实这次实现编译器的过程中，虽然设计很花费时间，但是最后发现自己在调试和细节处理上花费的时间远远超出了设计所花费的。正如一些政治书上所说的，理论付诸于实践并不是一件容易的事情。
- 我的本研做的内容和 Java 内部的结构有一定的关系，其中也用到了一些 Java ASM 和 JVM 相关的工具，它们都是基于 Accept+Visitor 的编程模型的，也涉及到了一些和三地址码或者对象内存结构的相关的内容，所以此次编译实习对我本研的研究也有帮助。
- 总的来说我对自己的编译器实现效果非常满意。虽然有些瑕疵和错误的存在，但是当我真的整合完五步提交代码的时候，发现自己竟然真的写出了一个编译器的时候，还是有一些自豪的。我觉得这种成就感也是一大收获吧

建议

- 此次课程基本按照 UCLA 的原课进行，所提供的工具也很实用，不过我认为可以适当发挥学生的自由度。比如如果有些同学可以不通过五步而用更少的阶段实现，也应该给予鼓励。而不是完全按照 15*5 的方式评分。
- 这学期由于种种原因没有实现网络编程和测试，不过这不是很严重的问题，助教评分也挺辛苦的，我认为测试时可以提前公开一部分数据，测试完毕后可以公布每个人的错误用例。至少让我们知道自己错在哪里了。
- 最后其实我觉得这门课可以采取如 Web 技术概论或者各大 MOOC 课程形式的全在线授课，毕竟我觉得上课讲的东西和 PPT 也差不多，讨论和答疑也均可以在网上进行