学　　号 1400012962

编译原理实习报告

院　　系：信息科学技术学院

专　　业：计算机系

姓　　名：丁菲菲

二〇一七年一月

摘　　要

《编译实习》这门课程的主要任务是完成一个 Minijava 的编译器。设计过程有一个较高的起点：JavaCC 和 JTB 已经对 Minijava 的词法和语法进行了分析，由此构造出了一棵语法树，我们仅需在已有语法树的基础上进行编译器的实现。主要分为六个基本步骤：

(1) Minijava 类型检查

(2) Minijava 语言翻译为 Piglet 语言

(3) Piglet 语言翻译为 Spiglet 语言

(4) Spiglet 语言翻译为 Kanga 语言

(5) Kanga 语言翻译为 MIPS 语言

(6) 将 1～5 步串联起来，形成一个完整的 Minijava 编译器

本实验的实验环境如下：

（1）硬件环境：Mac OS X

（2）软件环境：Eclipse 、IntelliJ

目　　录

摘要 **I**

1 Minijava 类型检查 **1**

1.1 Minijava 概述 1

1.2 符号表设计 1

1.3 符号表构建与重复定义检查 3

1.4 剩余错误检查 4

1.5 错误打印与测试 4

2 Minijava 语言翻译为 Piglet 语言 **7**

2.1 Piglet 概述 7

2.2 符号表构建 7

2.3 翻译过程 9

2.4 测试 10

3 Piglet 语言翻译为 Spiglet 语言 **11**

3.1 Spiglet 概述 11

3.2 翻译过程 11

3.3 测试 12

4 Spiglet 语言翻译为 Kanga 语言 **13**

4.1 Kanga 概述 13

4.2 翻译过程 14

4.2.1 流程图建立 14

4.2.2 活性区间分析 15

4.2.3 寄存器分配 17
4.2.4 翻译映射 17
4.3 测试 17

5 Kanga 语言翻译为 MIPS 语言 **19**
5.1 MIPS 概述 19
5.2 翻译 20
5.3 测试 20

6 整合串联 **21**

7 心得与建议 **22**

致谢 **23**

1 Minijava 类型检查

1.1 Minijava 概述

Minijava 作为 Java 语言的一个子集，具有 Java 面向对象语言的基本特点，有类、继承、覆盖、多态等高级概念。但相比 Java，Minijava 更加简单易懂。基本特性如下:

（1）MiniJava 是 Java 的子集，缺省约定遵从 Java

（2）不允许方法重载

（3）类中只能申明变量和方法（不能嵌套类）

（4）只有类，没有接口，有继承关系（单继承）

（5）不支持注释

（6）一个文件中可以声明若干个类

（7）共支持 9 种表达式

（8）共支持 9 种基本表达式

1.2 符号表设计

对 Minijava 进行类型检查的第一步便是构建一个层次清晰的符号表。用它来收集符号属性，把它作为上下文语义的合法性检查的依据以及目标代码生成阶段地址分配的依据。

综合考虑了老师给的参考模型和实际需要，我的符号表结构如图 1.1 所示。符号表中所有的类都继承自 MyType 类，以实现之后操作上的一些方便。

（1）MyClasses 是一个总的类，存储所有的类，用一个 HashMap 以类名为 Key 维护所有的类。插入类、查找类、判断类是否重复等操作在 MyClasses 类中实现。

（2）MyClass 是一个存储类的结构，用 HashMap 维护类的成员变量以及成员函数结点。插入变量、查找变量、判断变量是否重复、插入方法、查找方法、判断方法是否重复等操作在 MyClass 中实现。此外，MyClass 还需要存储父类节点，以

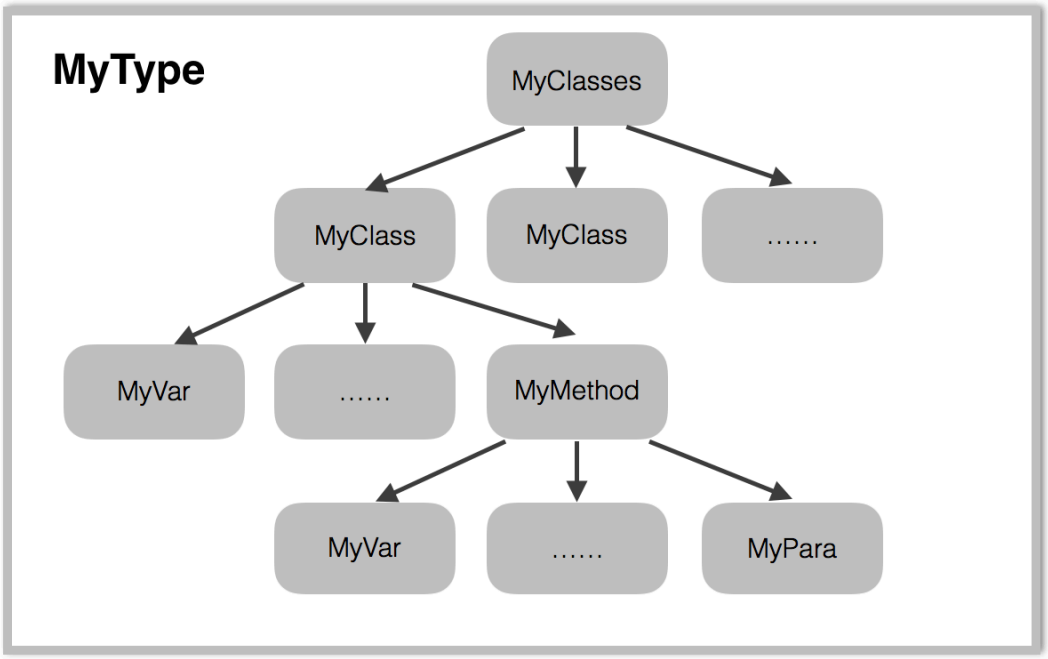

图 1.1　符号表结构

正确实现查找变量、查找方法：从当前类开始，查找对应的变量或者方法，然后往父类、父类的父类等逐步查找，若找到，返回第一个有效值，停止查找；否则报错。这样有效地实现作用域的判断。

（3）MyMethod 是一个存储方法的结构，用 HashMap 维护方法的参数和局部变量。插入参数变量、查找参数变量、插入局部变量、查找局部变量、判断局部变量是否重复等操作在 MyMethod 中实现。值得注意的是，查找变量这一操作比在 MyClass 中更为复杂：首先需要先检查参数变量，再检查局部变量，然后以方法所属类的成员变量、方法所属类的父类的成员变量、方法所属类的父类的父类的成员变量等顺序依次检查，直到找到为止，否则报错。判断局部变量是否重复这一操作也需要注意：不仅仅要检查局部变量，还需要检查参数变量。此外，还需要存储方法的返回类型、方法所属的类以方便后续的检查和操作。

（4）MyVar 是一个简单的存储变量的结构，仅需要存储变量的类型、所属方法或者类等基本信息。为实现一些更高级的检查，还可以维护两个布尔变量，分别代表是否初始化、是否使用过。

（5）MyPara 是一个存储参数变量的结构，类似 MyVar，此处不再赘述。

除了图 1.1 中所示的类，我还维护了 MyIdentifier 和 MyTypeName 两个类来存

储其他必要的信息，它们都继承自 MyType。其中，MyIdentifier 用来存储程序中出现的各种应该能在符号表中找到的变量名。MyTypeName 用来存储诸如 “3+2” 等表达式的形式，以实现访问者模式中自上而下的信息传递。

1.3 符号表构建与重复定义检查

在构建符号表的过程中可以同时进行重复定义检查，即第一次遍历语法树的过程中，在进入类、方法块时，对出现的类、方法、变量等进行重复定义检查并按 1.2 节的符号表设计分别插入到对应的结构体中。Main 函数中对应的入口代码如下：

```
// 建立符号表，检查重复定义的错误，传入BuildSymbolTableVisitor
root.accept(new BuildSymbolTableVisitor(), my_class);
```

重复定义包括：(1) 类名重复 (2) 同一类中变量名重复，允许子类的变量名覆盖父类的变量名 (3) 同一类中方法名重复，允许子类的方法覆盖父类的方法，但是如果参数类型列表对应相等且返回值类型不同，则报错 (4) 同一方法中变量名重复。

符号表构建与重复定义检查时机如下：

MainClass	插入主类（查重），构造main方法插入主类，并为该main方法插入一个参数
ClassDeclaration	插入类（查重）
ClassExtendDeclaration	插入类（查重），定义父类，若没有则缺省为“None”）
VarDeclaration	插入变量（查重），定义类型、所属域
MethodDeclaration	插入方法（查重），定义返回类型、所属域
FormalParameter	插入参数（查重）
ArrayType/BooleanType/ IntegerType	构造MyTypename类，自下而上传递类型
Identifier	构造MyIdentifier类，自下而上传递变量名

图 1.2 符号表构建与重复定义检查时机

在插入的同时检查是否重复定义，如果重复定义，则报错并取消插入操作，继续向后遍历。

1.4 剩余错误检查

第一遍扫描完成了符号表构建和对重复定义这一错误的检查。在第二遍扫描中完成对剩余错误的检查。错误如下：

（1）类的循环继承

（2）使用了未定义的类、方法或者变量

（3）赋值操作左右类型不匹配

（4）布尔操作的算子非布尔型，算数操作的算子非整型

（5）数组下标非整型

（6）打印操作的参数非整型

（7）方法调用参数不匹配

针对以上错误，具体检查时机如图 1.3 所示。

1.5 错误打印与测试

完成两次遍历后，程序中的错误已经完全检查出来了。在检查出错误的同时立刻将错误打印出来。我设计了 PrintMsg 这样一个类专门用来打印错误信息。具体代码大致如下：

```
public class PrintMsg {
    public static Vector<String> errors = new Vector<String>();
    public static void printMsg(int line, int column, String error_msg){
        String msg = "Line " + line + " Column " + column + ": " + error_msg;
        errors.addElement(msg);
    }
    public static void printAll() {
        int sz = errors.size();
        for (int i = 0; i < sz; i++)
            System.out.println(errors.elementAt(i));
    }
}
```

ClassExtendDeclaration	1、检查所继承的父类是否存在，若不存在，则报错 2、检查是否有循环继承，如果有，则报错
MethodDeclaration	1、如果参数类型列表对应相等且返回值类型不同，则报错 2、检查返回值与返回类型是否匹配，如果返回普通的数据类型，则直接检查是否一致，若为自定义的类名，则需检查返回值是否属于返回类型的子类
Type	若从下层的得到了理应在符号表中存在的变量，但是不存在，则报错
AssignmentStatement	1、检查左值在符号表中是否存在 2、检查左右类型是否一致 3、若类型不一致，则检查右边类型是否是左边类型的子类
ArrayAssignmentStatement/ ArrayLookup/ArrayLength/ ArrayAllocationExpression	1、检查数组变量的类型是否为数组 2、检查数组的下标是否为整型 3、检查右边类型是否为整型
IfStatement/ WhileStatement/ AndExpression/ NotExpression	检查算子是否为布尔型
MessageSend	1、判断调用者是否为类 2、判断方法是否在类中存在 3、判断参数列表是否匹配
PrimaryExpression	判断大部分的未声明变量
AllocationExpression	判断类操作是否为类名，并构造返回类型为对应的类

图 1.3　错误检查时机

通过这种方式可以定位到错误所在的位置，具体到行数和列数。当然，如果仅仅以通过测试样例为目的，那么仅仅需要维护一个布尔变量，初始化为 true。当发现错误的时候，将布尔变量置为 false，并停止遍历，输出 “Type error”；两次遍历结束后若布尔值仍为 true，表明无错，输出 “Program type checked successfully”。

2 Minijava 语言翻译为 Piglet 语言

2.1 Piglet 概述

Piglet 是一种接近中间代码的语言，但不是严格的三地址码。采用后缀式表达，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。[1] 整个 Piglet 代码由一块一块的方法块组成，与 Minijava 的主要区别如下：

（1）Piglet 中取消了类的概念，转而将类展开为一块一块的方法块

（2）Piglet 中取消了变量名的概念，转而用一系列 TEMP 变量替代。

（3）对于 Minijava 中的 new 操作，Piglet 需要显式地开辟与存取内存。

2.2 符号表构建

不难发现，Minijava 代码翻译为 Piglet 代码最重要的一步是要展开类。因此，在 Minijava 类型检查中构建的符号表在这里自然就失效了。在构建新的符号表之前，通过对 Piglet 的语法分析，我们不难发现类中变量的调用是通过一个一次间接指针得到实际地址，类中函数的调用是通过一个二次间接地址得到实际地址。结构图如图 2.1 所示。

考虑到继承问题，在一个类中，就变量而言，需要将该类中所有变量、该类的父类中的所有变量、该类的父类的父类的所有变量等依次添加到该类中，若有同名变量，不覆盖；就方法而言，需要将该类中所有方法、该类的父类中的所有方法、该类的父类的父类的所有方法等依次添加到该类中，若有同名方法，根据参数判断是否覆盖。

根据 Piglet 结构，我们可以通过深度优先遍历 Minijava 代码语法树来构建符号表。构建符号表和下一步翻译可以在一次遍历中完成。

每个类维护两个 HashMap 来存储该类的变量和方法的信息。同时还需要维护

[1] 刘先华老师课堂 PPT

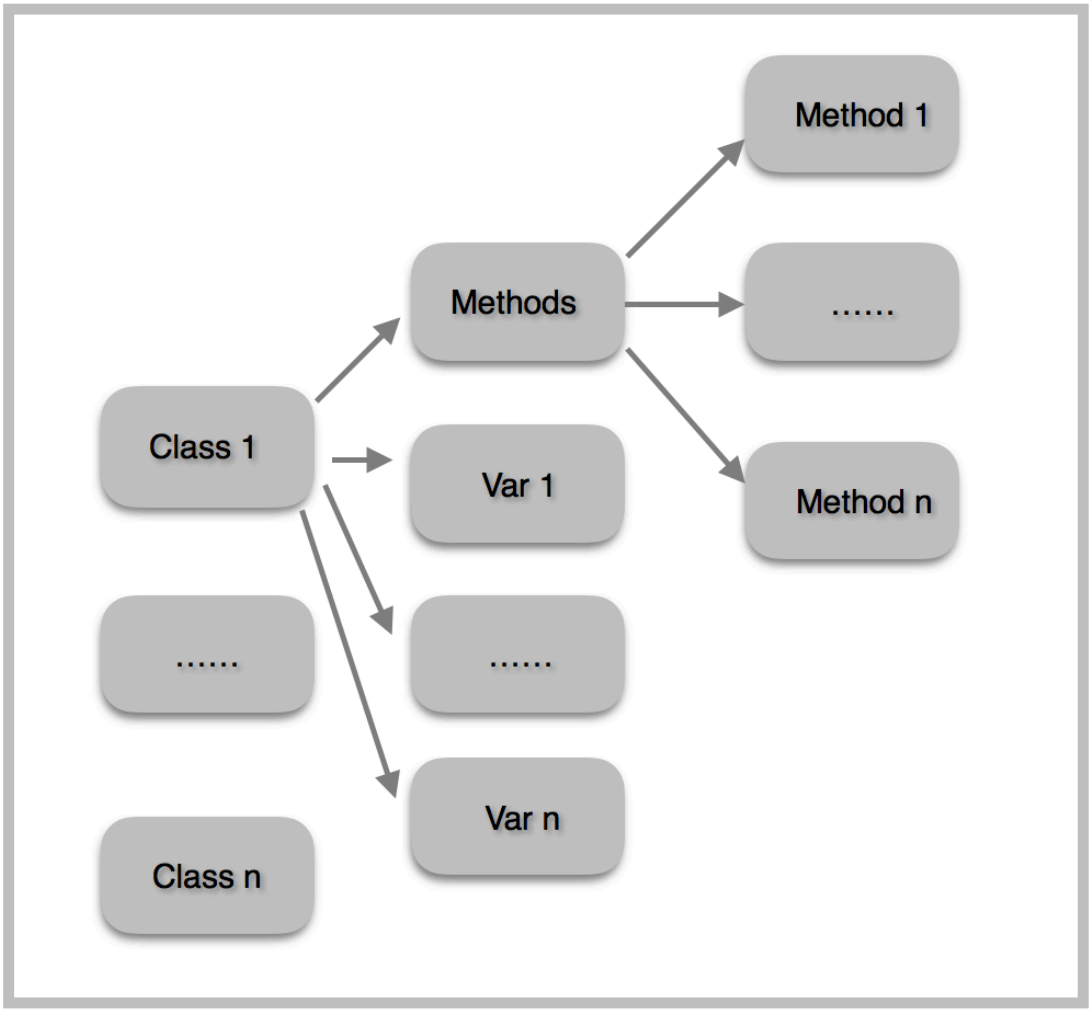

图 2.1　Piglet 结构图

两个全局 HashMap 来存储所有的变量和方法的信息。如下代码是对每个类分别创建变量列表:

```
public Vector<String> buildVarMap(MyClass m_class){
    Vector<String> v;
    // 如果该类已经创建过了，直接返回变量列表
    if (varMap.containsKey(m_class.name))
        return varMap.get(m_class.name);
    // 如果该类有父类，首先添加父类的变量列表
    if (all_classes.GetClass(m_class.father) != null)
        v = (Vector<String>)
        buildVarMap(all_classes.GetClass(m_class.father)).clone();
    else
        v = new Vector<String>();
```

```
    // 将该类自定义的变量加入变量列表
    Iterator<Map.Entry<String, MyVar>> iter =
        m_class.mj_var.entrySet().iterator();
    while (iter.hasNext()){
        Map.Entry entry = (Map.Entry)iter.next();
        String key = (String)entry.getKey();
        v.add(key);
    }
    varMap.put(m_class.name, v);
    return v;
}
```

类似这样，我们就可以得到了对每次 new 一个类所需要开辟的空间大小，也就是其变量数加上一个方法指针。

2.3 翻译过程

翻译与符号表构建可以在同一次遍历中完成。从深度优先遍历的角度来看，对每个节点，仅需要做好本身的翻译工作，子节点的翻译工作交给子节点完成，子节点完成后将翻译结果返回给父节点，父节点汇总后统一返回给上层父节点或者直接输出。以在我看来难度较大的数组分配（ArrayAllocationExpression）为例：

```
/**
 * f0 -> "new"
 * f1 -> "int"
 * f2 -> "["
 * f3 -> Expression()
 * f4 -> "]"
 */
public MyType visit(ArrayAllocationExpression n, MyType argu) {
    int t1 = cur_tmp++;
    int t2 = cur_tmp++;
    int t3 = cur_tmp++;
    int t4 = cur_tmp++;
    int l1 = cur_label++;
    int l2 = cur_label++;
    PrintPiglet.pBegin();
    PrintPiglet.p("MOVE TEMP " + t1 + " ");
    n.f3.accept(this, argu);
```

```
    PrintPiglet.pln("");
    PrintPiglet.pln("MOVE TEMP " + t2 +
    " HALLOCATE TIMES 4 PLUS 1 TEMP " + t1);
    PrintPiglet.pln("HSTORE TEMP " + t2 + " 0 TEMP " + t1);
    PrintPiglet.pReturn();
    PrintPiglet.pln("TEMP " + t2);
    PrintPiglet.pEnd();
    return null;
}
```

在 ArrayAllocationExpression 中，我们无需在本节点中计算子节点 Expression 的值，而是直接调用 Expression 节点返回的结果，将它存入 t1。在本节点中，我们需要完成的是数组空间分配的任务，即将 Minijava 中的 new 语句转换为对应的 Piglet 语句。对于数组，需要开辟一块数组大小加 1 的内存空间，将大小存入 t2 中。额外多出来的一块用来存放数组的长度。对于其他语句，参照 Piglet 语法逐步翻译即可。

此外，对于参数个数大于 20 的情况也是翻译的一个难点。因为 Piglet 最多只能使用 TEMP19 来传参，所以这里需要为第 19 个之后的所有参数开辟一个数组，将它们存入一个数组，并将数组的地址存入 TEMP19 中。类似如下处理：

```
PrintPiglet.pln("MOVE TEMP " + t1 + " " + nParaLeft);
PrintPiglet.pln("MOVE TEMP " + t2 + " HALLOCATE TIMES 4 PLUS 1 TEMP " + t1);
PrintPiglet.pln("HSTORE TEMP " + t2 + " 0 TEMP " + t1);
```

2.4 测试

运行自己的程序，在标准输入中输入 Minijava 程序，将自己的程序所翻译输出的 Piglet 代码另存成一个文件，然后用命令行 cd 到 pgi 解释器所在的目录，运行

```
java PgInterpreter < your_pg_file
```

若运行结果与 Minijava 的运行结果一致，则代表该样例通过。

3 Piglet 语言翻译为 Spiglet 语言

3.1 Spiglet 概述

Spiglet 的主要特点是去除了嵌套表达式，使其与三地址代码更为接近。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 语句可以使用表达式，print 语句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

(3) 表达式只有四种类型，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)

3.2 翻译过程

根据 Spiglet 语法，我们主要的任务是将嵌套多层的复合语句层层剥离，加入中转的 TEMP，从而翻译成一条条的简单语句。类似翻译到 Piglet 的过程中的深度遍历，在一条语句中需要用临时单元替换表达式时，先进入到表达式中进行计算和输出，然后表达式根据需要返回简单表达式或复杂表达式。以 HSTORE 语句为例：

```
/**
 * f0 -> "HSTORE"
 * f1 -> Exp()
 * f2 -> IntegerLiteral()
 * f3 -> Exp()
 */
public String visit(HStoreStmt n, MyType argu){
   flag = 1;
   String temp = n.f2.accept(this, argu);
   flag = 0;
   System.out.printf("HSTORE " + n.f1.accept(this, argu) + temp +
      n.f3.accept(this, argu));
   System.out.println();
```

```
    return null;
}
```

语句中 Expression 语句交由对应的子节点处理，父节点仅需要将子节点反馈的结果存起来，最后一并输出即可。

此外，本次翻译需要两次遍历。因为 Piglet 和 Spiglet 语言都需要用到 TEMP，为避免序号混淆，第一次遍历用来统计 Piglet 程序中使用的 TEMP 的数量。这样在第二次遍历过程中，我们就可以在这个值的基础上自增取 TEMP 号。用一个全局变量 nNum 维护即可：

```
/**
 * f0 -> "TEMP"
 * f1 -> IntegerLiteral()
 */
public MyType visit(Temp n, MyType argu){
    MyType _ret=null;
    n.f0.accept(this, argu);
    int temp = Integer.parseInt(n.f1.f0.toString());
    if (temp > nNum)
        nNum = temp;
    n.f1.accept(this, argu);
    return _ret;
}
```

3.3 测试

运行自己的程序，在标准输入中输入 Piglet 程序，将自己的程序所翻译输出的 Spiglet 代码另存成一个文件，然后用命令行 cd 到 spgi 解释器所在的目录，运行

```
java SpgInterpreter < your_spg_file
```

若运行结果与 Minijava 的运行结果一致，则代表该样例通过。

4 Spiglet 语言翻译为 Kanga 语言

4.1 Kanga 概述

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 24 个寄存器：

a0-a3：存放向子函数传递的参数

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

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

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

（4）Call 指令的格式发生较大的变化，没有显式调用参数，需要通过寄存器传递，没有显式 “RETURN”。a0-a3 存放向子函数传递的参数，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 是从 1 开始的，即 PASSARG i 需要用 SPILLEDARG i-1 访问

例如，考虑对一个某过程 P 的调用，参数已经放在寄存器 t1, t2, t3, t4, t5 中，返回值需要放在 t6 中：

```
MOVE a0 t1 // 首先将4 个参数放到“a”寄存器中
MOVE a1 t2
MOVE a2 t3
MOVE a3 t4
PASSARG 1 t5 // 将 t5 放栈中（注意是”1”，不是”0”）
CALL P
MOVE t6 v0 // 返回值在 v0 中
```

（5）过程的头部含 3 个整数（例如：procA [5] [3] [4]）

第一个整数的含义与 Spiglet 相同，表示参数个数第二个整数用于表示过程需要的栈单元个数，包含参数（如果需要）、溢出（spilled）单元、需要保存的寄存器

第三个整数是过程体中的最大参数数目，例如，如果 procA 调用 procB 时用 3 个参数，而调用 procC 用 2 个参数，调用 procD 用 4 个参数, 那么这个整数设为 4 [2]

4.2 翻译过程

根据对 Kanga 语法的分析，我们不难发现翻译的重点在于栈操作和寄存器分配，我的翻译过程依次为流程图建立、活性分析、寄存器分配、翻译映射，需要遍历三次。Main 函数如下所示：

```
public class Main{
    public static void main(String[] args){
        try{
            new SpigletParser(System.in);
            Node root = SpigletParser.Goal();
            //存储过程名和过程块的映射
            HashMap<String, FlowGraph> proc_map = new HashMap<String,
                FlowGraph>();
            //存储每个Label的名字和对应的行数
            HashMap<String, Integer> label_map = new HashMap<String,
                Integer>();
            root.accept(new FlowGraphNodeVisitor(proc_map, label_map));
            root.accept(new FlowGraphVisitor(proc_map, label_map));
            AssignRegister assign_register = new AssignRegister(proc_map);
            assign_register.assignRegister();
            root.accept(new Spiglet2KangaVisitor(proc_map));
        }
        ......
    }
}
```

4.2.1 流程图建立

考虑到实现效率和代码可读性，我将每一条语句作为一个基本块构建结点，对每一个过程单独画流图。每个语句结点的结构如下所示，每个结点需要维护自己的行号、前驱块、后继块、左值、右值等信息：

```
public class FlowGraphNode {
```

[2] 刘先华老师课堂 PPT

```
    public int line;
    public HashMap<Integer, FlowGraphNode> before_nodes;
    public HashMap<Integer, FlowGraphNode> after_nodes;
    public HashSet<Integer> use;
    public HashSet<Integer> def;
    public HashSet<Integer> in;
    public HashSet<Integer> out;
}
```

每个过程块的结构如下所示，每个流图中需要维护过程块的名字、行号、参数个数、栈单元个数、过程体最大参数个数、过程块中的结点等基本信息：

```
public class FlowGraph{
    public String name;
    public int line;
    public int para_num;
    public int stack_num;
    public int max_para_num;
    public HashMap<Integer, FlowGraphNode> nodes;
    public Vector<FlowGraphNode> vNode;
    public HashSet<Integer> call_pos;
    public HashMap<Integer, Register> tmp_map;
    public HashMap<Register, Integer> regT;
    public HashMap<Register, Integer> regS;
    public HashMap<Register, Integer> stack;
}
```

在第一次遍历中，完成对结点的构建，并将结点添加到对应的流图中，即以行号为 key 添加到 HashMap 类型的 nodes 中。第二次遍历则相对比较复杂，需要对流程图中的每一个结点填充它的前驱块、后继块、左值、右值等信息，同时还要记录下过程参数大小、过程体最大参数个数。对于栈单元个数，需要在寄存器分配完成后才能确定。

4.2.2 活性区间分析

这里为了对活性区间做准确的分析，引入了 IN 与 OUT 的概念，意思就是在该结点入口处或出口处活跃。主要根据《编译原理》龙书上的活性分析的公式进

行设计:

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

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

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

算法并不复杂，如下：

```
public void livenessAnalysis(){
    flag = 1;
    while(flag == 1){
        flag = 0;
        for (int i = proc_cur.vNode.size() - 1; i >= 0; i--){
            FlowGraphNode node_cur = proc_cur.nodes.get(i);
            for (Map.Entry<Integer, FlowGraphNode> entry2:
                node_cur.after_nodes.entrySet())
                    node_cur.out.addAll((entry2.getValue()).in);
            HashSet<Integer> new_in = new HashSet<Integer>();
            new_in.addAll(node_cur.out);
            new_in.removeAll(node_cur.def);
            new_in.addAll(node_cur.use);
            if(!new_in.equals(node_cur.in)){
                node_cur.in = new_in;
                flag = 1;
            }
        }
    }
}
```

活性分析完毕后可以得到每个 TEMP 的活性区间，由于一个 TEMP 的活性区间可能会断成若干截，我将第一截的起始点作为活性区间的开始，将最后一截的结束点作为该 TEMP 活性区间的终点，也就是说正好覆盖这若干截。可以看到，这是一种很保守的思路。

4.2.3 寄存器分配

寄存器分配算法可谓是该阶段中最复杂的部分了，我采用了线性扫描算法。

对于一般的 TEMP （非参数、非返回值），它们存放的位置不外乎三处：t 寄存器，s 寄存器和栈。根据代价，我优先将它们放入 t 寄存器中，其次放入 s 寄存器中，但这样代价比 t 寄存器大，因为需要对栈做一次存取操作。当 t 寄存器和 s 寄存器都不够用，只能将它放到栈上，这意味着每次调用这个值，都需要对栈进行存取操作，外加一个寄存器中转操作，代价最大。

还有一种特殊的 TEMP，它的活性区间跨越了 CALL 语句，意味着我们需要保证它的值在函数调用之后仍然是不变的。我在这里将它放到被调用者保存寄存器中，即从 s 寄存器开始分配，当 s 寄存器不够用，则将它放到栈上。

是放入 s 寄存器，t 寄存器还是栈上，这里就需要用到线性扫描算法。伪代码如图 4.1 所示（见下页）。

4.2.4 翻译映射

完成了寄存器分配工作，最后一次遍历便是将输入翻译为 Kanga 语言了。遇到每个 TEMP 则根据它是被分配到寄存器还是栈上，进行相应的转换即可。至于过程的第二个参数，只需在上一部分配寄存器时，将该过程中所用到的 s 寄存器数，加上被 SPILL 的额外 TEMP 数，再加上多于 4 个的参数数即可。对于参数个数超过 4 个的过程，要以 PASSARG 的形式传参，额外的参数都要到栈上去取。此外，进入过程之前和之后，要额外从栈上存取 s 寄存器，以保证它的值在过程调用前后都相同。

4.3 测试

运行自己的程序，在标准输入中输入 Spiglet 程序，将自己的程序所翻译输出的 Kanga 代码另存成一个文件，然后用命令行 cd 到 kg 解释器所在的目录，运行

```
java KgInterpreter < your_kg_file
```

若运行结果与 Minijava 的运行结果一致，则代表该样例通过。

```
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
```

图 4.1　线性扫描算法

5　Kanga 语言翻译为 MIPS 语言

5.1　MIPS 概述

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。MIPS 与 Kanga 最大的区别就是在于 MIPS 需要手动地构造运行栈并对栈进行存取操作。因此，我们仅需要维护三个指针：$fp，$sp，$ra。我采用了 MIPS 规范栈的布局，如图 5.1 所示：

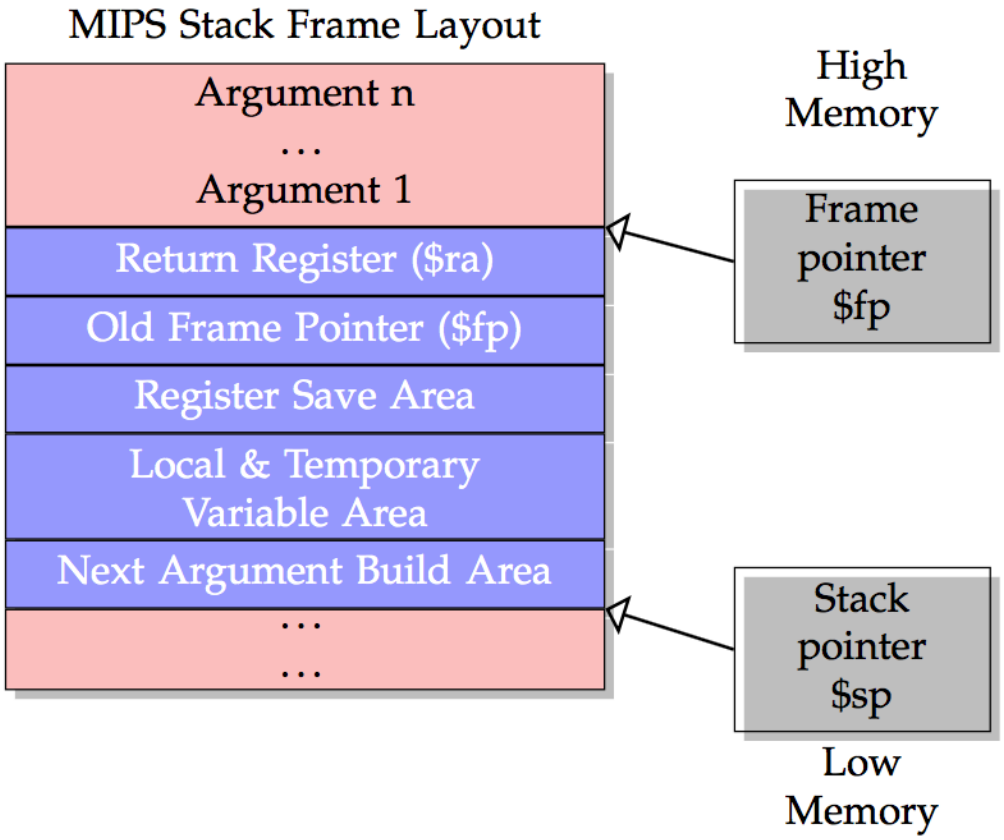

图 5.1　MIPS 栈布局

5.2 翻译

这一阶段中，即使对 MIPS 语言没有太多了解，对栈的结构把握得不是很清晰，仍然可以根据样例找规律般地翻译成功。这一阶段的翻译类似 Piglet2Spiglet，甚至比它还要简单。需要注意的是要仔细观察样例的处理，以防遗漏，具体翻译细节不再赘述。

5.3 测试

本阶段的测试方式和之前有所不同，在标准输入中输入 Kanga 程序，将程序输出的代码保存为另一个文件后，用 Spim 模拟器运行。若运行结果与 Minijava 的运行结果一致，则代表该样例通过。

6 整合串联

最后一阶段的任务是将前五个阶段整合串联起来，即输入 Minijava 程序，输出可在 Spim 上运行的 MIPS 程序。在这里，我将前一个阶段的输出缓存为一个文件，再将这个文件作为下一个阶段的输出，最终在控制台输出 MIPS 程序。整合阶段找到了之前阶段的很多 bug，花费了不少时间进行排查。

7 心得与建议

看到自己一个阶段一个阶段慢慢完成的编译器终于正确运行的那一刻是很有成就感的。回顾每一个阶段，随着代码越来越复杂，感觉到自己的 coding 和 debug 能力在不断提升。当然，最大的好处是再次验证了“实践是检验真理的唯一标准”这句话，仅有理论课的学习是远远不够的，还需要这样一门实习课，来锻炼我们的动手能力，提高对理论的认识水平。

最后，提一些小小的建议：

（1) 就我本人而言，这是我第一次写 Java 代码，而且是第一次用 Java 将 Java 语言的子集 Minijava 翻译为我更加不了解的 MIPS 语言 (没有修过《计算机组成》这门课)。虽然后来逐步发现整个过程并没有我想象得那么困难，但是还是希望老师能够提供类似 C 语言的可选方案，或者给出 Java 语言（尤其是 Java 语言中访问者模式）的教程。在自学了 Java 后还是对访问者模式不甚了解，和部分同学讨论后，大家也没有什么思路，以致于第一次作业花费了很长时间、走了很多弯路才稍稍摸清了其中的门道，其实弄懂之后才发现并不那么难懂。

（2）算是承接上一条建议吧，建议老师将第一次编写计算器的小作业改为或者扩充为针对 Java 访问者模式的简单练习，这样可以帮助我们对之后的 Minijava 类型检查快速上手。

（3）每一阶段都是在假设前几个阶段都正确完成的基础上进行的，比如翻译 Spiglet 阶段，我们参照着给出的几个 Spiglet 样例进行翻译，但实际上我们前一阶段自己生成的 Spiglet 代码与样例也许有很大不同，代码在样例上可以通过，但是在我们自己生成的 Spiglet 代码上就会出错。最后串起来的过程就会很麻烦，因为难以定位问题究竟出在哪一阶段。所以建议每一阶段不仅仅要完成本来的任务，还需要串起来，即依次为：Minijava2Piglet，Minijava2Spiglet，Minijava2Kanga，Minijava2MIPS。

致　　谢

感谢刘先华老师和高铮助教一学期的辛苦付出，感谢王子辰、马荣等同学课后对我的极大帮助，感谢所有和我一起选了这门课的同学，遇到你们，是一件幸运的事情！