2019 编译实习实验报告

盛楷文[*1] and 张晨滨[†1]

[1]EECS of Peking University

June 21, 2019

Contents

[*]1600012947@pku.edu.cn

[†]1600012901@pku.edu.cn

1 概述

1.1 编译器

简单讲，编译器就是将 “一种语言（通常为高级语言）” 翻译为 “另一种语言（通常为低级语言）” 的程序。一个现代编译器的主要工作流程：源代码 (source code) 预处理器 (preprocessor) 编译器 (compiler) 目标代码 (object code) 链接器 (Linker) 可执行程序 (executables)。

1.2 关于该程序

该程序的参考资料为UCLA CS 132 Project，目标是实现一个小型的 MiniJava 编译器。

- 输入：符合 MiniJava 语言规范的源程序
  MiniJava 是 Java 的子集，不允许方法重载和嵌套类（不允许存在名字相同的方法，类中只能申明变量和方法），不支持注释...
- 输出：能在 MIPS 模拟器 SPIM 上运行的目标代码
  SPIM 的输入是 MIPS 汇编语言程序，而不是机器语言程序
- 借助自动工具完成词法和语法分析 (JavaCC和JTB)
- 主要完成以下关键步骤：
  类型检查，中间代码生成，寄存器分配，映射机器指令

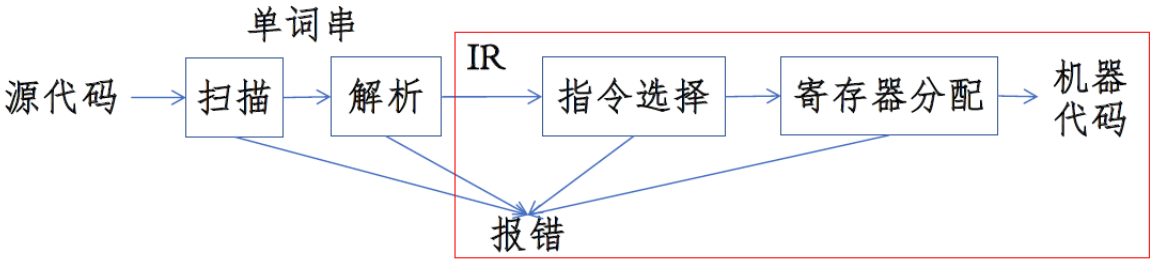

Figure 1: 关键步骤

1.3 程序源码

该程序的源码，测试数据，批处理程序可以从Github获得，具体操作详见该目录下 README.md。

1.4 编译器特点

1. 使用 on the fly 的翻译特色
2. 使用 Java visitor 编程模式构造符号表
3. 使用 JTB 和 JavaCC 作为词法和语法分析器前段实现

2 编译过程

2.1 整体思路

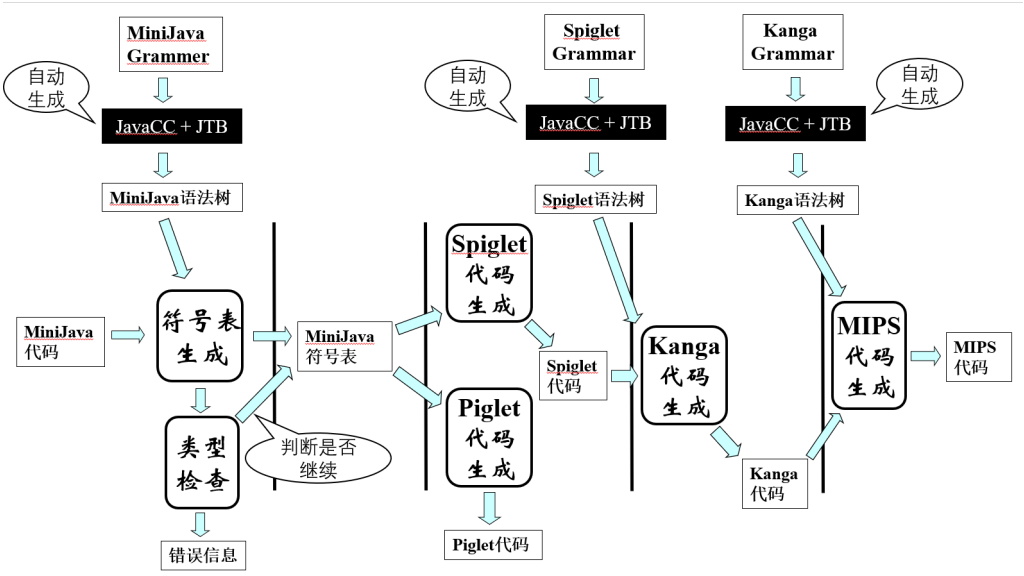

Figure 2: 总体流程

2.2 符号表构建

2.2.1 构建工具

利用 JTB 生成的 DepthFirstVisitor，按照深度优先顺序调用 Vistor 遍历语法树，并按照层次构建出符号表

2.2.2 主体结构

符号表中定义了以下类: MType MArray MInt MBool MClass MUndefined MScope MBlock MMethod MExpr MPrimExpr MVar

类的继承关系如下：

具体的代码组织如下所示：

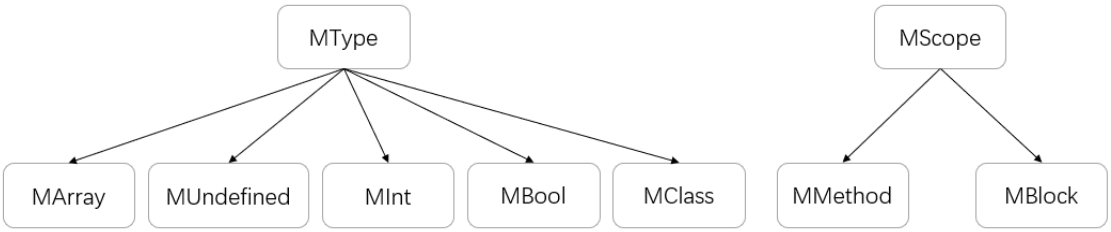

Figure 3: 继承

2.3 类型检查

2.3.1 流程

类型检查可以分为两个阶段：建立符号表、回填和错误检查。

1. 建立符号表：解析源程序，自顶向下遍历 JavaCC 和 JTB 构造的语法分析树，调用各个类的构造函数添加实例，最终逐步构建自己的符号表。
2. 回填：严格上来说应该也属于符号表建立过程，回填的目的是鉴于某些属性或变量的类型是源程序中的某个 Class，但该 Class 并未被及时解析到，所以需要在第一步遍历完语法分析树后再为这些特殊变量指定好正确的变量类型。回填函数在相应类中是名为 fillBack() 的方法。
3. 错误检查，自顶向下遍历符号表，逐层调用 register() 方法进行检查错误。

符号表的建立过程只使用了 JavaCC 生成的 DepthFirstVisitor，即深度优先搜索，我们的程序只为该 visitor 定义了处理 MainClass, ClassDeclaration 和 lassExtendsDeclaration 的 visit 函数，当 visitor 处理到以上三种节点时，会调用 MClass 的构造函数，而该构造函数会遍历当前节点的所有子孙节点直至构造出完整的符号表。

2.3.2 符号表的结构

每个类的代码主题结构及定义如下：MType 是所有类型 (MArray, MBool, MClass, MInt, MUndefined) 的父类，抽象表达一个类型应当具有的属性：

1. getName() 类型的名字
2. getSize() 类型占用的内存空间大小
3. isAssignable(MType target)

该类型的变量是否可以赋予 target 类型的值

```
public abstract class MType {
    abstract public String getName();
    abstract public int getSize();
    abstract public boolean isAssignable(MType target);
}
```

MArray, MBool, MInt 是最简单的三个类型，它们仅仅实现 MType 定义的方法即可，没有扩展出自己的特色方法。在我们的程序中，这三个 Class 都只有一个实例，存储在 SymbolTable 当中用于引用。

```
public class MArray extends MType {
    public String getName();
    public int getSize();
    public boolean isAssignable(MType target);
}
public class MBool extends MType {
    public String getName();
    public int getSize();
    public boolean isAssignable(MType target);
}
public class MInt extends MType {
    public String getName();
    public int getSize();
    public boolean isAssignable(MType target);
}
```

MClass 较为复杂，我们对 MClass 的理解是：每一个 MClass 的实例都是一个新的类型，所以每当有一个新的 Class 被声明，我们的都会创建一个对应的 MClass 实例并存储到 SymbolTable 当中用于引用。

```
public class MClass extends MType {
    // 符号表相关的属性
    private String name_;
    private String father_name_;
    private MClass father_;
    private int size_;
    private Boolean is_main_class;
```

```
// 当前类具有的方法和属性
// 各用一个ArrayList存储名字，一个HashMap存储实例
private ArrayList<String> all_methods_;
private HashMap<String, MMethod> methods_;
private ArrayList<String> all_vars_;
private HashMap<String, MVar> vars_;
// 转换成面向过程时method table和var table的构建
private HashMap<String, Integer> method_offset_;
private HashMap<String, Integer> var_offset_;
// 所有父类及祖先类
private ArrayList<MClass> father_list_;
// 符号表构造
private void parseMethod(NodeListOptional method_list) {}
public MClass(Node node) {}
// 回填阶段
public void fillBack() {}
// 类型检查阶段
// 检查循环继承
public void registerFather() {}
// 检查重复定义方法，并分布到每个方法中进行类型检查
public void registerMethod() {}
// 检查变量的重复定义
public void registerVar() {}
// 检查类时的方法调用关系
public void register() {
    registerMethod();
    registerVar();
}
public String getName();
public int getSize();
public boolean isAssignable(MType src) {}
// 构建var table和method table
public void createView() {}
// 翻译成piglet/spiglet过程中访问var table和method table
```

```
        public int queryMethodOffset(String method_name) {}
        public int queryVarOffset(String var_name) {}
        // 翻译成piglet
        public String generatePigletNewClassCode() {}
        // 翻译成spiglet
        public String generateSpigletNewClassCode() {}
    }
```

MUndefined 正是回填时需要处理的类型，一旦某个属性或变量被声明为一个用户自定义 Class 的实例，则在第一步符号表构建阶段，这个属性或变量的 type 会先暂时被认定为 MUndefined，之后的回填阶段会逐个将 MUndefined 类型替换为正确的 MClass 实例。

```
    public class MUndefined extends MType {
        // 记录MClass实例的名字
        private String class_name_;
        public MUndefined(String class_name) {}
        public String getClassName() {}
        public String getName() {}
        public int getSize() {}
        public boolean isAssignable(MType target) {}
    }
```

MScope 是一个抽象类，它是 MMethod 和 MBlock 的父类，之所以把 MMethod 和 MBlock 归入 MScope 是因为它们都可以视为代码块（作用域），我们在 MScope 当中添加了几个静态方法被 MMethod 和 MBlock 实例调用。

```
    public abstract class MScope {
        // 增加子块
        public abstract void addBlock(MScope scope) {}
        // 父类块
        public abstract MScope getFather() {}
        // 查询当前块及子块中的变量
        public abstract MVar queryVar(String var_name) {}
        // 符号表构建
        public static MBlock getBlock(NodeChoice choice, MScope father) {}
        public static void parseStatement(NodeListOptional statement_list,
        MScope father) {}
```

```
    public static void parseStatement(Statement statement, MScope
    father) {}
}
```

MMethod 继承自 MScope

```
public class MMethod extends MScope {
    // 符号表相关属性
    private MType ret_type_;
    private String name_;
    private HashMap<String, MVar> params_;
    private HashMap<Integer, String> index2name_;
    private HashMap<String, MVar> vars_;
    private ArrayList<MBlock> blocks_;
    private MClass owner_;
    private MScope father_;
    private MExpr return_;
    // 符号表构建
    private void parseParam(NodeOptional param_list) {}
    public MMethod(MClass owner, Node node) {}
    public MMethod(MClass owner, MainClass class_node) {}
    public void addBlock(MScope block) {}
    // 回填函数
    public void fillBack() {}
    // 检查方法
    public void register() {}
    // 处理方法重写问题：判断两个方法的参数是否匹配
    public boolean matchParam(ArrayList<MExpr> exprs) {}
    public boolean matchParam(MMethod method) {}
    // 生成Piglet代码
    public String generatePigletMethodCode() {}
    // 生成Spiglet代码
    public String generateSpigletMethodCode() {}
}
```

MBlock 继承自 MScope，较为特殊的是，为了处理方便，我们把每个语句都视为一个独立的

MBlock。

```
public class MBlock extends MScope {
    // 符号表相关属性
    private int which_;
    private MScope father_;
    private ArrayList<MBlock> children_;
    private MExpr expression_, index_expression_;
    private String var_name_;
    private MVar var_;
    // 构建符号表
    public void addBlock(MScope block) {}
    public MBlock(MScope father, NodeChoice node_choice) {}
    // 检查方法
    public void register() {}
    // 生成Piglet代码
    public String generatePigletBlockCode(int tab, boolean write) {}
    // 生成Spiglet代码
    public String generateSpigletBlockCode(int tab, boolean write) {}
}
```

MVar 是变量在符号表的抽象表达

```
public class MVar {
    // 符号表相关属性
    private MType type_;
    private String name_;
    private boolean assigned_ = false;
    private int length_;
    private MClass class_owner_;
    // 分配得到的Temp id
    private int pigletTempID;
    private int spigletTempID;
    // 符号表构造方法
    public MVar(MType type) {}
    public MVar(String param_name_) {}
```

```
    // 检查是否使用了未分配的数组或类实例
    public void allocate(MExpr expr) {}
    // 为当前变量分配Temp id
    public int getPigletTempID() {}
    public void setPigletTempID(int tempID) {}
}
```

MExpr 是表达式在符号表上的反映

```
public class MExpr {
    // 符号表相关属性
    private MScope father_;
    private int which_;
    private String op_;
    private MPrimExpr prim_expr_, prim_expr2_;
    private ArrayList<MExpr> exprs_;
    private String method_name_;
    private MMethod method_;
    private MType type_;
    // 构造符号表
    public MExpr(Expression expr, MScope father) {}
    // 检查方法
    void register() {}
    // 生成Piglet代码
    public String generatePigletExpressionCode(int tab, boolean write) {}
    // 生成Spiglet代码
    public String generateSpigletExpressionCode(int tab, boolean write) {}
}
```

SymbolTable 负责获取输入文件，是符号表的本体，并提供公用的符号表构建所需方法，之后的几个任务中也有类似的 SymbolTable，功能类似，在下文中也不再提及 SymbolTable。

```
public class SymbolTable {
    // syntaxtree的根
    private static Goal root_;
    // 输入文件的相关信息
    private static String file_name_;
```

```
    private static File file_;
    // 主类
    private static MClass main_class_;
    // 其他类
    private static ArrayList<MClass> class_list_;
    // 存储各个MType的实例
    private static HashMap<String, MType> type_map_;
    // 构造syntaxtree
    public static boolean parse(final File file) {}
    // 符号表构造相关的方法
    public static void addMainClass(MClass c) {}
    public static void addClass(MClass c) {}
    public static void buildClass() {}
    public static MType getType(Type t) {W}
    public static MType getType(String type_name) {}
    public static boolean parseVar(NodeListOptional var_list, HashMap<String,
    public static MClass queryClass(String className) {}
    public static ArrayList<MClass> getClassList () {}
}
```

2.4 MiniJava to Piglet

Minijava 转向 Pigelet 的基本思路是使用 Minijava 的符号表信息，进行相应的转换，直接生成 Piglet 代码。由于 Piglet 是一种面向过程的语言，所以需要在对 Minijava 这种面向对象的语言进行相应的封装。

下面将对类的初始化，过程变量与标号翻译，参数溢出进行详细介绍。

2.4.1 类的初始化

我们对于每一个类（主类）除外进行了封装，这个原因是基于来自 UCLA 的官网上的代码，在处理 new 一个新的变量的时候非常的繁琐，而且可读性较差，于是我们将 new 抽象封装成一个单独的函数。每当执行 new 语句的时候，会直接调用该函数。

类的初始化的难点在于如何构建方法表和变量表，然后在实际的函数调用和变量访问的时候都使用指针的形式进行操作。在生成 new 的函数代码之前，我们对于每个类执行 createView 函数，为每个函数先生成方法表和变量表，具体的组织结构如下图所示：

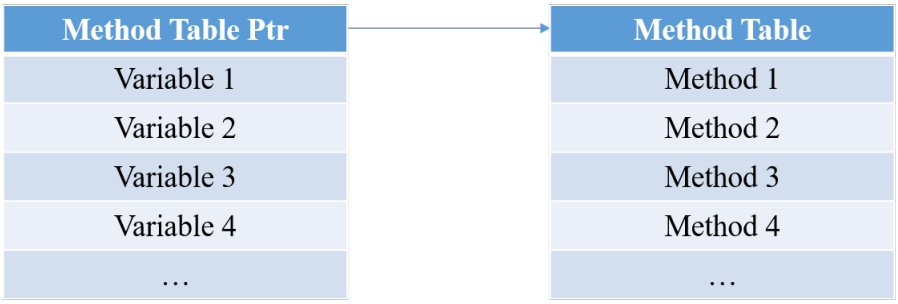

Figure 4: 方法、变量表

可以看到，在类的成员表中的第 0 个元素存储的是指向方法表的内存地址。从第 1 个元素开始，存储的成员变量的地址。我们在 Minijava 的符号表中为每一个方法和变量构建了一个 HashMap，记录变量名到偏移量的映射。在构建类的成员表的时候，最大的难点在于方法的覆盖和属性的隐藏。

对于属性而言，由于属性是隐藏的，所以不需要考虑覆盖的操作，在当前类访问的同名变量，将是类的层次结构中最下层的变量，所以对于变量在子类中的存储，我们住需要按照从父类到子类的顺序，按顺序存储变量即可。届时访问查找变量的时候，自底向上进行查找，便能够得到需要的正确的变量。

对于方法而言，就会更加复杂一些，需要进行覆盖。同样我们按照父类到子类的顺序进行扫面，依次将方法加入到方法表中，如果在搜索到子类的时候发现了同名方法，那么直接替换掉方法表中已经放入的父类方法即可。这样在实际访问方法的时候，同样采用自底向上的方式进行搜索，搜索到的方法一定是正确的，因为保证了父类所有的同名方法已经被覆盖掉了。

2.4.2 过程变量与标号翻译

由于 Piglet 是一种中间代码形式，不需要考虑寄存器分配以及和寄存器的上线。所以我们采用了一种很暴力的方式，所有使用新的寄存器的地方的编号、以及 label 的标号都是不相同的，使用一个全局变量进行更新即可。

2.4.3 参数溢出

Piglet 中对于函数的参数个数上限做了限制，只允许最多有 20 个参数以寄存器的形式进行传递，剩下的参数必须放在内存中传递。我们为了保险起见，0-17 个参数将直接放在寄存器中进行传递，从第 18 个参数开始，将单独开辟一块内存，放置参数的地址，最后将参数表的起始地址以变量的形式传递到函数中去，便可以解决参数溢出的问题。

2.4.4 具体实现

除了 new 代码之外，其余的代码生成全部使用 on the fly 的增量翻译方式，可以减少一些内存的占用存储中间代码。具体的实现如下，在 MClass 中使用 generateSpigletNewClassCode 生成初始

化类的封装代码，在方法，表达式，初级表达式中都使用 generatePigletMethodCode 进行相应的代码翻译。

```
public String generateSpigletNewClassCode();

public String generatePigletMethodCode();
```

2.5 Piglet to Spiglet

从 Piglet 到 Spiglet 的转换想对简单，因为两者的基本语法是相似的，只有在几个细节上稍有不同。最大的不同点在于 Spiglet 不允许使用任何的嵌套表达式，只用 move 可以使用表达式，print 可以使用初级表达式。即便如此，对于 Piglet 的修改并不大，所以我们没有采用重新构建 Piglet 的 Visitor 来实现这一转换，我们直接在 Piglet 的代码基础上进行了小范围的修改，将上述的嵌套代码进行拆分即可，没有很特殊的操作。

2.6 Spiglet to Kange

Spiglet2Kange 的实现流程为：符号表构建、基本块构建、活跃变量分析、寄存器分配、代码生成

2.6.1 符号表构建

类似构造 minijava 的符号表的方式，符号表的元素一共涉及四类：MProcedure, MStmt, MExp, MSimpleExp。Mprocedure 对应 Spiglet 中的 Procedure, 每个 Spiglet 程序都是由一个 Main Procedure 和零个或多个 Procedure 构成，因而我们的符号表整体上也是由多个 MProcedure 构成。Spiglet 中的 Procedure 又由多个 Stmt 构成主体，因此我们的 MProcedure 也包含多个 Stmt 作为主体信息。MStmt 代表 Stmt 的同时还将其多个分支包含进去，它有一个 which_ 属性，指示当前的 MStmt 实例代表那种 Stmt。每个 Stmt 可能含有 Temp、Label、IntegerLiteral、Exp、SimpleExp，MStmt 存储整数表示 Temp，字符串表示 Label，整数表示 IntegerLiteral，还包含了 MExp 和 MSimpleExp，MExp 以及 MSimpleExp 的符号表实现逻辑类似 MStmt。符号表元素之间没有继承关系

```
public class MStmt {
    // 构建符号表相关属性
    private int which_;
    private int tmp_id_, tmp_id2_, integer_;
    private String pre_label_, label_;
    private MExp exp_;
    private MSimpleExp sexp_;
```

```
    private MProcedure procedure_;
    // 寄存器分析相关属性
    private HashSet<Integer> used_ids_, defined_ids_;
    // 符号表构造
    public MStmt(NodeSequence node_seq, MProcedure procedure) {}
    // 寄存器分析
    public HashSet<Integer> getDefinedIds() {}
    public HashSet<Integer> getUsedIds() {}
    // 转换为Kange需要的方法
    public String getLabel() {}
    public boolean isJump() {}
    public String getExtraSuccessor() {}
    public getParamNum() {}
    public String toKanga(
        HashMap<Integer, Integer>, HashSet<Integer> OUTs, int stack_num) {}
}

public class MExp {
    // 符号表相关的属性
    private int which_, tmp_id_;
    private String op_;
    private ArrayList<Integer> tmp_ids_;
    private MSimpleExp sexp_;
    // 构造符号表
    public MExp(Exp exp) {}
    // 寄存器分析
    public HashSet<Integer> getUsedIds() {}
    // 转换为Kange需要的方法
    public int getParamNum() {}
    // 专门为Call准备的解析方法
    public String getCall(
        HashMap<Integer, Integer> tmp2reg, int stack_num) {}
    // 解析到当前Exp之前需要先设置好寄存器
    public String prepare(
```

```
        HashMap<Integer, Integer> tmp2reg, int stack_num) {}
    // 转换为Kanga代码
    public String toKanga(HashMap<Integer, Integer> tmp2reg) {}
}

public class MProcedure {
    // 符号表相关的属性
    public static String registers_[];
    private String label_;
    private int param_num_;
    private HashMap<String, MStmt> label2stmt_;
    private ArrayList<MStmt> stmt_list_;
    private MSimpleExp return_exp_;
    private HashMap<MStmt, Integer> stmt2index_;
    // 寄存器分析相关属性
    private final static int save_reg_num_;
    private ArrayList<ArrayList<Integer>> successors_;
    private ArrayList<ArrayList<Integer>> predecessors_;
    private ArrayList<HashSet<Integer>> INs_;
    private ArrayList<HashSet<Integer>> OUTs_;
    private HashMap<Integer, HashSet<Integer>> edges_;
    private int spill_cnt_;
    private HashMap<Integer, Integer> tmp2reg_;
    private int max_param_num_;
    // 构造符号表
    private void formStmtList(NodeListOptional stmts) {}
    public MProcedure(Goal goal) {}
    public MProcedure(Procedure procedure) {}
    // 寄存器分析
    private void constructMap() {}
    // 转换为Kanga代码
    private void getMaxParamNum() {}
    public String toKanga(boolean store} {}
}
```

```
public class MSimpleExp {
    // 符号表相关的属性
    private int which_;
    private int tmp_id_, integer_;
    private String label_;
    // 构造符号表
    public MSimpleExp() {}
    public MSimpleExp(SimpleExp simple_exp) {}
    // 寄存器分析
    public int getUsedId() {}
    // 转换为Kanga代码
    public String prepare(
        HashMap<Integer, Integer> tmp2reg, int stack_num) {}
    public String toKanga(HashMap<Integer, Integer> tmp2reg) {}
}
```

2.6.2 基本块构建

较为简单，我们将每个 MStmt 都视为一个基本快，MProcedure 的构造函数在构造好所有 Mstmt 的结构后，会调用 constructMap() 函数构建基本快之间的联系方式：前驱和后继关系。结果会存入 successors_ 和 predecessors_ 两个 ArrayList<ArrayList<Integer>>，index 表示 MStmt 标号，如 successors_[0] 表示第 0 个 MStmt 的所有后继 Mstmt 的编号。

2.6.3 活跃变量分析

是编译技术课上一种寄存器分配和死代码消除算法，核心是维护基本快 IN, OUT, USE, DEF 集合。它们的定义如下：1) IN: 在基本块入口处活跃的变量集合 2) OUT: 在基本块出口处活跃的变量集合 3) USE: 在基本块中，未经定义就使用的变量集合 4) DEF: 在基本块中，定义的变量集合对于一个基本块来书，USE 和 DEF 都是常量，在代码中可以用 MStmt 的 getUsedIds() 和 getDefinedIds() 分别得到，具体实现并不困难，只需要找出当前 MStmt 所定义和使用的 Tmep id 即可。而 IN 和 OUT 集合则采用以下算法来计算：

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

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

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

      IN[n] = use[n] U (out[n] - def[n]);

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

Figure 5: 活跃变量分析

2.6.4 寄存器分配

采用图染色分配算法，结合 Kanga 寄存器的实际，确定寄存器语义如下：s0-s7: 通用，在函数调用上下文中保证值不变，由被调用者保存 t0-t9: 通用，在函数调用上下文中可能会被修改 a0-a3: 函数返回值 v1: 临时存放溢出变量

2.6.5 寄存器冲突图构建

完成了活跃变量分析之后，会构建寄存器冲突图，结果存入 HashMap<Integer, HashSet<Integer» edges_，作为 Key 值的 Integer 是 Temp Id，每个 Temp Id 在 edges_ 中会对应一个 HashSet，该 Set 存储的都是与当前 Temp Id 产生冲突的其他 Temp Id。判断两个 Temp Id 是否构成冲突的方法是查看它们是否同时出现在某个 IN Set 或者 OUT Set 当中。

2.6.6 图染色

流程如下（设可用寄存器有 k 个）：1) 找到一个度数小于 k-1 的节点，将其从图中删除并放入栈中 2) 如果找不到这样的节点，则选择一个度数最大的节点，将其溢出，并从图中删除 3) 执行 1) 和 2) 直到冲突图为空 4) 依次将节点弹出栈并染色，同时保证该节点的颜色和已染邻接节点的颜色不同

2.7 Kanga2MIPS

Kanga2MIPS 的实现流程为：构造符号表，转换为 MIPS。

2.7.1 符号表构建

符号表元素之间没有继承关系

```
public class MExp {
    // 符号表相关属性
    private int which_, reg_id_;
    private MSimpleExp sexp_;
    private String ops_[] = {"LT", "PLUS", "MINUS", "TIMES"};
    private int op_;
    // 对应MIPS的操作符
    private String mips_ops_[] = {"slt", "add", "sub", "mul"};
    // 构造符号表
    public MExp(Exp exp) {}
    // 作为操作数时，确定操作符是move li 还是 la
    public String getOperator() {}
    // 转换为MIPS代码时，需要预先处理HAllocate
    public String prepare() {}
    // 转换为MIPS
    public String toMIPS() {}
}
public class MSimpleExp {
    // 符号表相关属性
    private int which_, reg_id_, integer_;
    private String label_;
    // 构造符号表
    public MSimpleExp(SimpleExp simple_exp) {}
    // 作为操作数时，确定操作符是move li 还是 la
    public String getOperator() {}
    // 转换为MIPS
    public String toMIPS() {}
}
```

```
public class MProcedure {
    // 存储寄存器名字
    public static String registers_[];
    // 符号表相关属性
    private String label_;
    private int param_num_, stack_cell_num_, max_param_num_;
    private ArrayList<MStmt> stmt_list_;
    private HashMap<String, MStmt> label2stmt_;
    private HashMap<MStmt, Integer> stmt2index_;
    // 构造符号表
    private void formStmtList(NodeListOptional stmts) {}
    public MProcedure(Goal goal) {}
    public MProcedure(Procedure procedure) {}
    // 计算溢出参数个数
    public int getSpillParamCnt() {}
    // 调用当前过程时，有多少个参数被存放在栈中
    public int getUpStackParamCnt() {}
    // 转换为MIPS
    public String toMIPS(boolean is_main) {}
}

public class MStmt {
    // 符号表相关属性
    private int which_;
    private int reg_id_, reg_id2_, integer_;
    private String pre_label_ = null, label_ = null;
    private MProcedure procedure_ = null;
    private MExp exp_ = null;
    private MSimpleExp sexp_ = null;
    // 构造符号表
    public MStmt(NodeSequence node_seq, MProcedure procedure) {}
    public String getLabel() {}
    // 转换为MIPS
```

```
        public String toMIPS() {}
    }
```

2.7.2 栈结构分配

本次实现的核心是栈结构的安排，栈结构如下图所示：

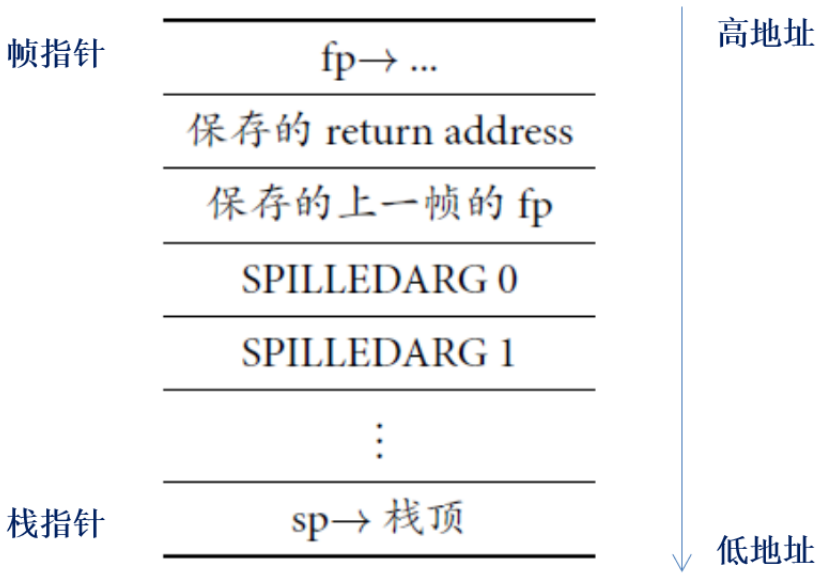

Figure 6: 栈结构

Kanga 代码中已经统一了 PASSARG 和溢出变量，因而可以直接通过 sp 加上偏移的方式访问到。子例程的实现包含以下操作：1. 操作栈，分配空间 2. 保存所有用到的 S 寄存器 3. 将所有用到的入参拷贝到 S 寄存器或 T 寄存或溢出空间里 4. 结束时恢复 S 寄存器和栈帧并且在整个代码的结尾加上大家都采用的 _print, _halloc, newl, str_er 用来实现 Kanga 的 PRINT, HALLOCATE 和 ERROR 即可。

3 编译器实现

3.1 工具软件

3.1.1 Eclipse

Eclipse 是一个开放源代码的、基于 Java 的可扩展开发平台。就其本身而言，它只是一个框架和一组服务，用于通过插件组件构建开发环境。幸运的是，Eclipse 附带了一个标准的插件集，包括 Java 开发工具（Java Development Kit，JDK）。Eclipse 可以在开发代码的过程中自动报错，正因为这点我们才优先选择 Eclipse 作为开发工具，而非 Visual Studio Code。

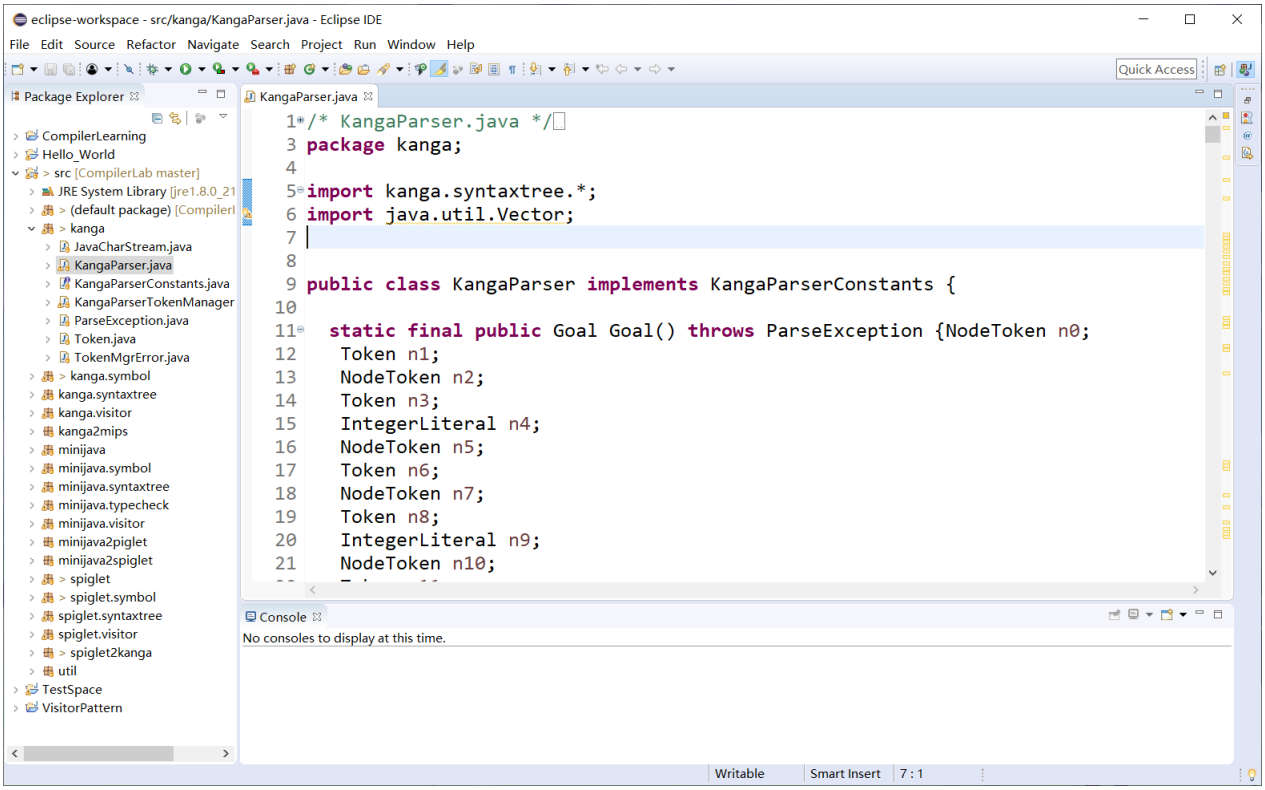

Figure 7: Eclipse

3.1.2 Visual Studio Code

该编辑器也集成了所有一款现代编辑器所应该具备的特性，包括语法高亮，可定制的热键绑定，括号匹配以及代码片段收集。界面较 Eclipse 美观，我们用这个软件辅助查看已经写好的代码、JavaCC JTB 生成的代码、编译器的中间输出结果如 Kanga 代码等。

3.1.3 JavaCC 和 JTB

我们采用的词法和语法分析器是 JavaCC 和 JTB。只需要使用 CS132 提供的.jj 文件即可构造语法分析树和 Visitor。在实际使用中，可以通过 Goal() 方法访问语法分析树的树根，从而能够顺着树根按照深度优先搜索的顺序构造出我们自己设计的符号表。以 MiniJava.jj 为例子，适用如下命令即可生成语法分析树和 Visitor：java -jar jtb132.jar minijava.jj java -cp javacc.jar javacc jtb.out.jj

3.1.4 Piglet Interpreter

这个是 Piglet 和 Spiglet 的解释器，将 Piglet 代码作为其输入即可得到运行结果。

3.1.5 Kanga Interpreter

这个是 Kanga 的解释器，将 Kanga 代码作为其输入即可得到运行结果。

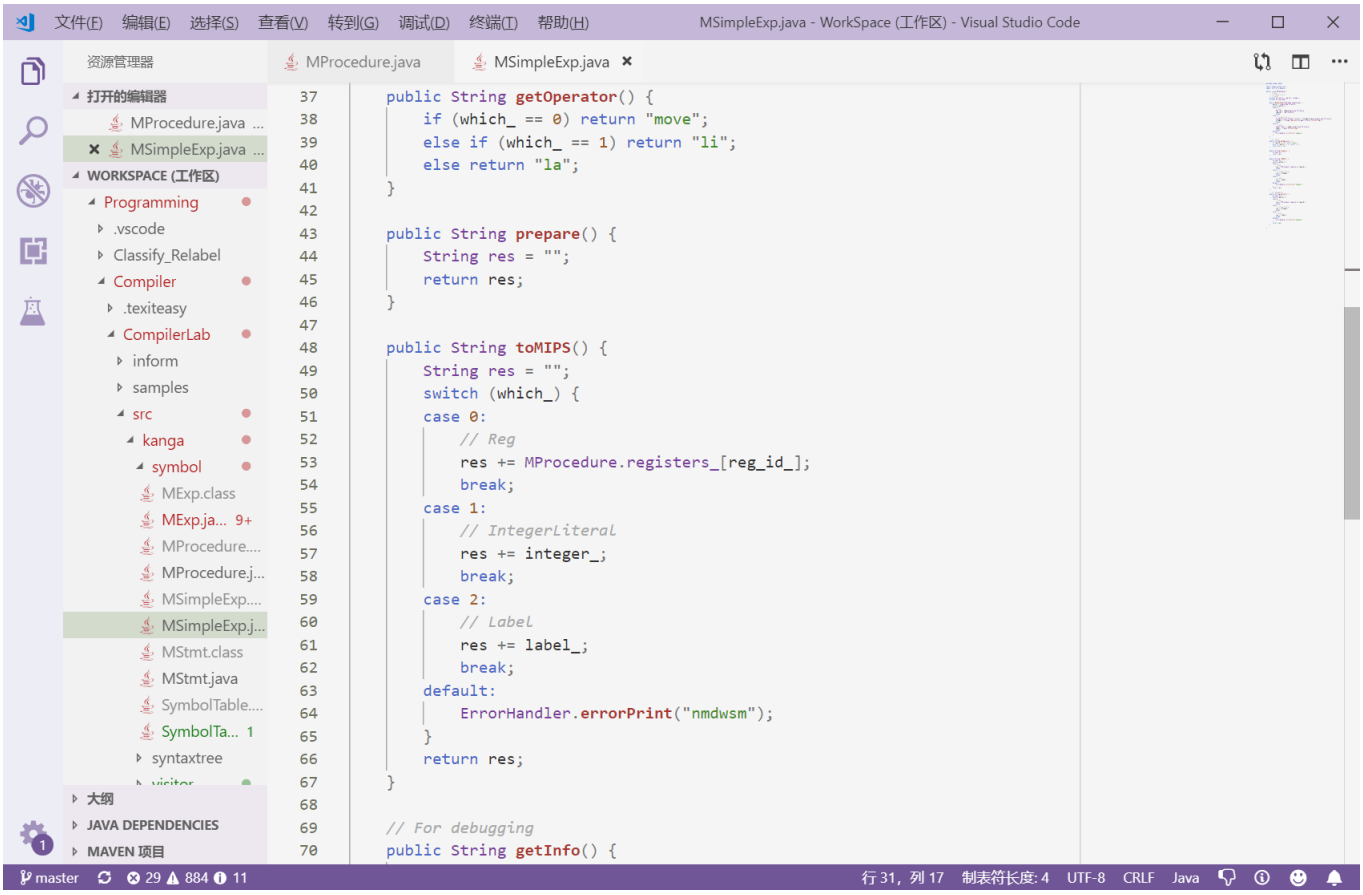

Figure 8: VSCode

3.1.6 QtSpim

这个是 Windows 系统下的 MIPS 模拟器，和之前几个解释器不同的是，该模拟器通过 UI 载入 MIPS 汇编代码得到运行结果。该模拟器 UI 界面操作非常不方便，且不支持命令行操作，无法批处理输出结果，给开发带来了很大的不便利。

3.2 各个阶段的编码细节

3.2.1 Visitor 的实现和使用

从 MiniJava 到 Spiglet 再到 Kanga，都仅定义了 ClassTreeBuilder 继承 DepthFirstVisitor。其中 MiniJava 的 ClassTreeBuilder 含有三个 visit 函数，分别用来访问 MainClass，ClassDeclaration，ClassExtendsDeclaration；Spiglet 只有一个 visit 函数访问 Goal；Kanga 也只有一个 visit 函数访问 Goal。以 Kanga 的 visit 访问 Goal 为例子，它会调用我们定义的 kanga.symbol.SymbolTable 类的 parse 方法处理该 Goal：遍历该 Goal 的所有 procedure 节点，并自顶向下调用 MProcedure, MStmt, MExp, MSimpleExp 的构造函数，构造出自己的符号表。另外两个语言的符号表构造类似。

3.2.2 生成另一种语言的操作

以 Kanga 为例子，在实现 Visitor 并自顶向下构造出符号表后，程序会再自顶向下遍历一次符号表，逐层调用 toMIPS 函数生成 MIPS 代码，并将结果拼接在一起输出到目标文件中即可。当然，在 MiniJava 转 Piglet 或 Spiglet 时，在构造符号表和生成代码的中间还要加入类型检查；在 Spiglet 转 Kanga 时，中间也要加入一个寄存器分配的过程。

3.3 测试

3.3.1 样例构造与测试方法

测试样例来源于 UCLA 网站上的标准程序，包括了 MiniJava 部分、sPiglet 部分和 Kanga 部分。测试时，使用到了 UCLA 提供的解释器或模拟器，或命令行运行，或通过 UI 界面运行，运行样例并检查输出的一致性。

3.3.2 测试中发现的错误

错误主要还是出现在从 Spiglet 转 Kanga 过程的测试中，主要包含以下两种情况：

1. Spiglet 出现某句 MOVE TEMP id Exp，但在接下来的语句中并没有使用该 id，如果该 TEMP id 和接下来要使用的某个 TEMP id_2 被分配到了同一个寄存器，那么计算过程是可能出错的，所以我在该语句的解析过程中加入了检查 id 是否处于当前 MStmt 的 OUT 数组当中，如果没有出现则将该语句输出为 NOOP。

2. 在寄存器分配的过程中，忘记将 return 语句当做一个 MStmt 进行考虑，导致 return 语句用到的 TEMP id 没有出现在之前语句的 OUT 数组中，从而因没有为该 TEMP id 分配寄存器而出错。

4 实习总结

4.1 编译器总结

基于编译技术课上学到的基础知识以及编译实习课上的额外指导，我们通过五个阶段的分步实现，最终完成了一个能够将 MiniJava 翻译为 MIPS 的编译器。我们实现的代码可读性较好，风格简洁。

4.2 收获与体会

理论与实践的结合至关重要，本次实习告诉了我们，在编译技术课上所学到的还远远不够，充足的理论并不能造就一名优秀的程序员，唯有将课程理论付诸实践才能真正掌握一门技术。与此同时，我们在探索编译器的过程中渐渐发现，编译器并非神秘而不可透视，通过一步步的实践探索，我们通过自己的努力成功写出了自己的编译器，给我们带来了十分充足的成就感。在本次编写编译器的过程中，我们也遇到了许多难题：例如寄存器的分配算法上，我们最开始认为线性扫描会比图染色算法更加容易，但是随着讨论的深入，我们一时无法相处如何应对分支结合问题，所以我们最终还是采用了图染色算法，结合之前在编译技术课程上的理解，通过自己的努力实现了该算法。

4.3 对课程的建议

我们在本次实习实现的主要是编译器的后端部分，词法和语法分析仅仅只是调用 JavaCC 和 JTB 实现。我们认为，编译器的前端设计也是十分重要的，也是需要花一定的功夫才能实现的部分。即使编译技术课上主要考察的内容是前端，但我们也仅局限于理论上的理解，缺乏实践终究是无法真正掌握前端设计的奥秘。因此，我们建议可以适当添加一些前端相关的小作业，增加同学们对编译器前端设计的掌握程度。

4.4 分工

4.4.1 类型检查

盛楷文：类型语法的检查

张晨滨：符号表构建

4.4.2 Minijava2Piglet

盛楷文：语法转换

张晨滨：符号表构建

4.4.3 Spiglet2Kanga

张晨滨：符号表构建活跃变量分析

盛楷文：寄存器分配 Kanga 代码生成

共同讨论：使用线性分配寄存器算法还是使用图染色分配寄存器算法的讨论。

4.4.4 Kanga2MIPS

张晨滨：MIPS 代码生成

盛楷文：构造符号表

共同讨论：栈空间的分配问题。

4.4.5 分工解释

为了让我们都能够熟悉编译器后端的搭建，我们在各个任务之间交替完成源代码的符号表构建和目标代码的生成。此外，较复杂的部分如面向对象转换为面向过程的存储空间安排、寄存器分配问题、栈空间分配问题都是由我们两人一起讨论，并合理分工实现的。