编译实习报告

从 MiniJava 到 MIPS 的编译器实现

刘佳凡 1400013007

2017 年 1 月

目录

1 简介 **4**

1.1 程序目标 4

1.2 实现步骤 4

1.3 MiniJava 简介 5

1.4 Visitor 设计模式 6

1.5 辅助工具 7

1.5.1 JavaCC 7

1.5.2 JTB 7

1.5.3 使用方法 7

1.6 参考资源 8

2 MiniJava 类型检查 **8**

2.1 MiniJava 语法 8

2.2 代码结构 11

2.3 实现步骤 12

2.4 符号表的建立 13

2.5 类型检查的实现 13

3 从 MiniJava 到 Piglet **14**

3.1 Piglet 简介 14

3.2 Piglet 语法 16

3.3 代码结构 17

3.4 实现步骤 18

3.5 数组的翻译 18

3.6 方法的翻译 19

3.7 控制流的翻译 19

3.8 类的翻译 19
3.8.1 封装 20
3.8.2 继承和多态 20
3.9 变量的翻译 21

4 从 Piglet 到 SPiglet **22**
4.1 SPiglet 简介 22
4.2 SPiglet 语法 22
4.3 代码结构 23
4.4 实现步骤 24
4.5 复合语句的消去 24

5 从 SPiglet 到 Kanga **24**
5.1 Kanga 简介 24
5.2 Kanga 语法 25
5.3 代码结构 27
5.4 实现步骤 29
5.5 流图的建立 29
5.6 活性分析与活性区间 30
5.7 寄存器分配算法 32
5.7.1 图着色算法 32
5.7.2 线性扫描算法 32
5.8 参考文献 33

6 从 Kanga 到 MIPS **34**
6.1 MIPS 简介 34
6.2 MIPS 语法 34
6.2.1 汇编信息 34

6.2.2 寄存器约定 34
6.2.3 MIPS 指令 35
6.2.4 运行栈 36
6.3 代码结构 36
6.4 实现步骤 36
6.5 运行栈的维护 37

7 总结 **37**
7.1 各模块的整合 37
7.2 测试结果 37
7.3 项目重点 38
7.4 课程收获 39
7.5 课程建议 39

1 简介

1.1 程序目标

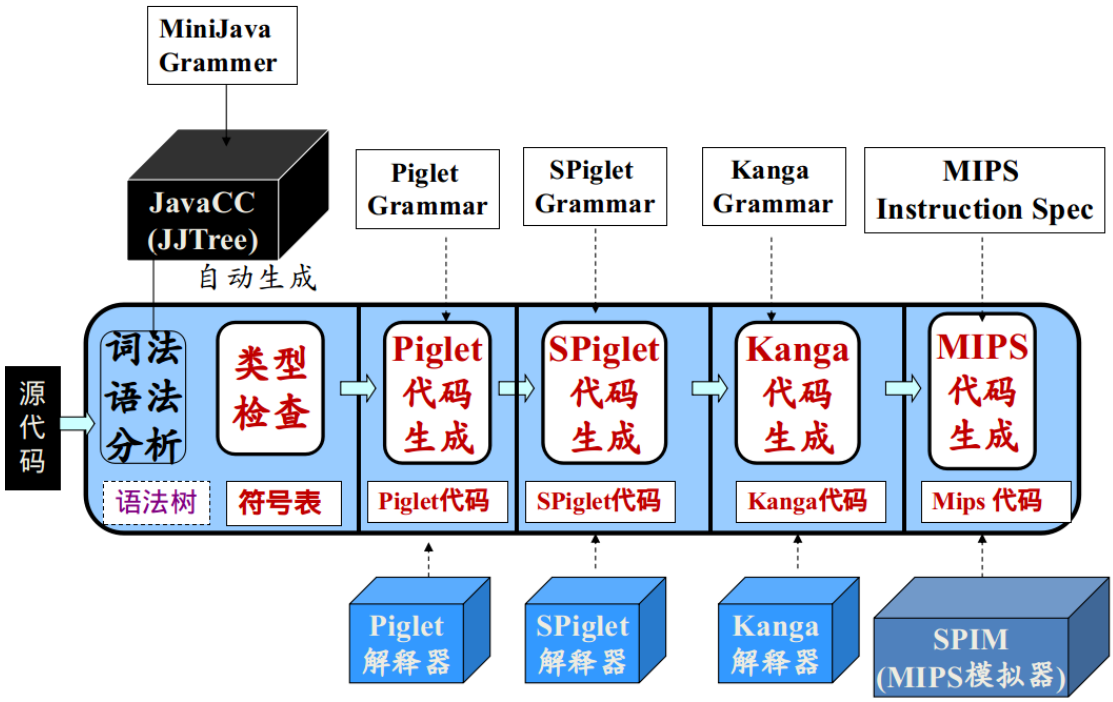
本实习项目由 UCLA CS132 Compiler Construction 课程设计。项目目标为使用 Java 语言设计一个将 MiniJava 编译为 MIPS 目标代码的编译器。MiniJava 是 Java 语言的一个简化后的子集。编译器的要求如下：

- **输入**：符合 MiniJava 语言规范的源程序 P
- **输出**：检查 P 的词法、语法、语义，对 P 进行编译，并开展优化，生成能在 SPIM 模拟器上运行的目标码
- **中间环节**：类型检查、生成基本中间代码、寄存器分配、映射为机器指令

1.2 实现步骤

编译过程分为以下五步：

- **MiniJava 类型检查**：建立符号表，通过类型检查对源码进行（浅）语义分析
- **从 MiniJava 到 Piglet**：将相对高层的类、数组、方法等语法翻译为底层语法
- **从 Piglet 到 SPiglet**：将上一步结果中一些复合迭代的语句简化
- **从 SPiglet 到 Kanga**：通过建立流图、活性分析，为变量实现寄存器的分配
- **从 Kanga 到 MIPS**：维护运行栈，最终翻译出符合 MIPS 规范的目标代码



以上步骤均建立在由 JavaCC 完成的的词法分析和语法分析、由 JTB 生成的支持 Visitor 模式的语法分析树的基础上。同时后四个步骤分别有 Piglet 解释器、SPiglet 检查工具、Kanga 解释器、SPIM 模拟器帮助检查编译的结果。后面将简单介绍这些工具。

1.3 MiniJava 简介

MiniJava 是 Java 的子集，缺省约定遵从 Java，主要区别如下：

- 不允许方法重载，但允许重写（支持多态）
- 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)

- 一个文件中可以声明若干个类，有且只有一个 Main 类，类不能申明为 public
- 表达式 (Expression) 包括：加、减、乘、与、小于、数组访问、数组长度、方法调用、基本表达式，省略了除、或、相等、大于等表达式，不支持连加
- 基本表达式 (Primary Expression) 包括：整数、true、false、对象、this、new 对象、new 数组、非、括号

1.4 Visitor 设计模式

Visitor 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。

Visitor 模式的实现方法是将代码划分为对象结构和一个 Visitor：每个类中包含一个以 Visitor 作为参数的 accept 方法，一个 Visitor 为每个类包含一个 visit 方法。Visitor 模式的特点包括:

- 可以方便地在对象中定义一个新的操作，而不用修改对象对应的类
- 不需要大量的重新编译，只需要重新编译 Visitor
- 通过类的层次结构进行访问，可以方便地进行层次化的遍历

将 Visitor 模式应用与此编译器的设计过程，可以大大简化程序的设计方法。在编译的每一个步骤中，我们可以通过实现不同的 Visitor 在抽象语法树上遍历实现不同的行为，而不需要修改语法树的内部定义。在使用 Visitor 的过程中，我们常在 visit 函数的递归过程中在抽象语法树中向上/向下传递一些必要的信息，对应了语法制导翻译中的综合属性/继承属性。

1.5 辅助工具

1.5.1 JavaCC

JavaCC（Java Compiler Compiler）是一个能自动生成词法分析器和语法分析器的程序。其使用方法如下：

- 输入：一个 JavaCC 语法文件 *.jj
- 输出：一个包含词法分析器和语法分析器的程序 (由多个 Java 文件组成)

1.5.2 JTB

JTB（Java Tree Builder) 类似于 JJTree，可以自动生成给定文法的语法分析树，采用 Visitor 设计模式。

- 输入：一个 JavaCC 语法文件 *.jj
- 输出：syntaxtree 和 visitor 两个目录以及文件 jtb.out.jj
  - syntaxtree 目录下的每个类代表抽象语法树的一种结点
  - visitor 目录包含了遍历抽象语法树所有结点的各种 Visitor 接口（编译的各个步骤都要实现新的 Visitor 对抽象语法树进行若干次遍历）
  - jtb.out.jj 可以使用 JavaCC 进一步生成支持抽象语法树的语法分析器

1.5.3 使用方法

以处理 MiniJava 为例，运行以下命令行指令：（“-p minijava” 参数可以指定所有生成文件在 minijava 包内，后续各个步骤放在不同的包中，便与管理）

```
java -jar jtb132.jar minijava.jj -p minijava
javacc jtb.out.jj
```

然后即可以在 Java 中使用自己改写的 Visitor 来遍历抽象语法树：

```
new MiniJavaParser(System.in);
Node AST = MiniJavaParser.Goal();
AST.accept(new MyVisitor());
```

1.6 参考资源

- 课程主页：UCLA CS132 Compiler Construction
  http://web.cs.ucla.edu/~palsberg/course/cs132/project.html
- JavaCC
  https://javacc.java.net/
- JTB
  http://compilers.cs.ucla.edu/jtb/

2 MiniJava 类型检查

2.1 MiniJava 语法

Goal	::=	MainClass (TypeDeclaration)* <EOF>
MainClass	::=	class Identifier ”{” ”public” ”static” ”void” ”main” ”(” ”String” ”[” ”]” Identifier ”)” ”{” (VarDeclaration)* (Statement)* ”}” ”}”
TypeDeclaration	::=	ClassDeclaration
	|	ClassExtendsDeclaration
ClassDeclaration	::=	class Identifier ”{” (VarDeclaration)* (MethodDeclaration)* ”}”
ClassExtendsDeclaration	::=	class Identifier ”extends” Identifier ”{” (VarDeclaration)* (MethodDeclaration)* ”}”

VarDeclaration	::=	Type Identifier ”;”
MethodDeclaration	::=	public Type Identifier ”(” (FormalParameterList)? ”)” ”{” (VarDeclaration)* (Statement)* ”return” Expression ”;” ”}”
FormalParameterList	::=	FormalParameter (FormalParameterRest)*
FormalParameter	::=	Type Identifier
FormalParameterRest	::=	, FormalParameter
Type	::=	ArrayType
	|	BooleanType
	|	IntegerType
	|	Identifier
ArrayType	::=	int ”[” ”]”
BooleanType	::=	boolean
IntegerType	::=	int
Statement	::=	Block
	|	AssignmentStatement
	|	ArrayAssignmentStatement
	|	IfStatement
	|	WhileStatement
	|	PrintStatement
Block	::=	{ (Statement)* ”}”
AssignmentStatement	::=	Identifier ”=” Expression ”;”
ArrayAssignmentStatement	::=	Identifier ”[” Expression ”]” ”=” Expression ”;”
IfStatement	::=	if ”(” Expression ”)” Statement ”else” Statement
WhileStatement	::=	while ”(” Expression ”)” Statement
PrintStatement	::=	System.out.println ”(” Expression ”)” ”;”
Expression	::=	AndExpression
	|	CompareExpression
	|	PlusExpression

	| MinusExpression
	| TimesExpression
	| ArrayLookup
	| ArrayLength
	| MessageSend
	| PrimaryExpression
AndExpression	::= PrimaryExpression "&&" PrimaryExpression
CompareExpression	::= PrimaryExpression "<" PrimaryExpression
PlusExpression	::= PrimaryExpression "+" PrimaryExpression
MinusExpression	::= PrimaryExpression "-" PrimaryExpression
TimesExpression	::= PrimaryExpression "*" PrimaryExpression
ArrayLookup	::= PrimaryExpression "[" PrimaryExpression "]"
ArrayLength	::= PrimaryExpression "." "length"
MessageSend	::= PrimaryExpression "." Identifier "(" (ExpressionList)? ")"
ExpressionList	::= Expression (ExpressionRest)*
ExpressionRest	::= , Expression
PrimaryExpression	::= IntegerLiteral
	| TrueLiteral
	| FalseLiteral
	| Identifier
	| ThisExpression
	| ArrayAllocationExpression
	| AllocationExpression
	| NotExpression
	| BracketExpression
IntegerLiteral	::= <INTEGER_LITERAL>
TrueLiteral	::= true
FalseLiteral	::= false
Identifier	::= <IDENTIFIER>

ThisExpression	::=	this
ArrayAllocationExpression	::=	new ”int” ”[” Expression ”]”
AllocationExpression	::=	new Identifier ”(” ”)”
NotExpression	::=	! Expression
BracketExpression	::=	(Expression ”)”

2.2 代码结构

以下代码位于 **minijava** 包中。

- typecheck
  - Main.java 主程序
  - ErrorMsg.java 格式化管理错误信息与对应的行号、列号，并可以分层次打印符号表
    * errorMsgs 储存所有错误信息的数组
    * verbose 控制错误输出的详细程度，分为三级：输出正确/错误；输出所有详细错误信息；输出详细错误信息与整个符号表的结构
- symboltable
  - MType.java 总抽象类，记录符号的基本信息，以下符号表中的所有类都继承此类
    * symbolName 符号名
    * lineNo 所在行号
    * colNo 所在列号
  - MClasses.java 包含所有类的最大符号表
    * classesTable 哈希表，存储所有类名到对应类映射
  - MClass.java 类的符号表
    * classesTable 保存最大符号表的引用，便于操作

    * superClassName 父类的类名，没有则为 null
    * memberVars 类的成员变量的符号表
    * memberMethods 类的成员方法的符号表
  - MMethod.java 方法的符号表
    * owner 方法所在的类
    * returnType 方法返回类型
    * params 方法的参数列表
    * otherLocalVars 方法的局部变量列表
    * varsTable 上述两种方法内变量的符号表
  - MVariable.java 变量的符号表
    * varType 变量类型
    * owner 变量的所属类/方法
    * init 是否被初始化（可用于不完整的数据流分析）
  - MParamList.java 参数列表的符号表
    * paramList 参数列表
    * owner 所在的方法

- visitor
  - MakeSymbolTable.java 建立符号表 Visitor
  - TypeCheck.java 类型检查 Visitor

2.3 实现步骤

步骤如下：

- 第一次 Visit，建立符号表，同时检查重复定义
- 符号表内即可以检查类的重载和循环定义
- 第二次 Visit，完成剩余类型检查

2.4 符号表的建立

建立符号表的重点在于保证符号表的完备性，并尽可能提高根据一个符号查找其他相关符号的便利性。我的符号表结构见上面代码结构部分。我没有将 int, boolean, int[], 自定义 class 类型四者分别分别包装为四个类，而是直接将它们统一作为 MType 类，遇到时根据符号名判断其对应的类型。

建立符号表的阶段我使用了以下的辅助函数：

```
// 将argu插入owner，返回是否插入成功
// argu为MClass/MMethod/MVariable类型，相应地owner为MClasses/MClass/
   MMethod或MClass类型。
public boolean insert(MType argu, MType owner);
```

2.5 类型检查的实现

需要检查的错误可以分为以下几类：

- **重复定义符号**

  在建立符号表过程中，要插入的符号已经存在

  包括重复定义了类、变量、方法，重载错误也包括在这一类（MiniJava 中子类只能重写父类方法而不能重载）

- **循环继承**

  符号表建立完成后，类的继承关系图存在环

  也可以在第二次 Visit 中对每个类分别向上追溯是否存在循环继承，复杂度稍高但实现简单

- **引用未定义符号**

  第二次 Visit 发现使用的符号不在符号表中

  包括使用未定义的类、变量、方法

- **不匹配**

  第二次 Visit 检查各种语句中的类型匹配情况

包括很多细节错误，几乎每个表达式都包含相应类型匹配的检查，在此不再赘述；除了类型匹配，同时还包括函数的参数个数的匹配问题

类型检查的阶段我使用了以下的辅助函数：

```
// 检查argu是否在owner内存在
public MType query(MType argu, MType owner);
// 检查subClass是否是superClass的子类（包括间接子类）
public boolean isSubClass(MClass subClass, MClass superClass);
// 查询owner内部的argu符号（可能是变量或类型）的类型名是否与type匹配
    （如果argu的类型名为type类的子类，也为正确匹配）
public boolean isMatched(MType argu, MType owner, String type);
// 检查left=right的等式能否匹配（匹配包括left为right的间接父类）
public void checkMatched(MType left, MType right, MType leftOwner,
    MType rightOwner);
// 以下分别检查argu是否为int/boolean/int[]类型
public boolean checkInt(MType argu, MType owner);
public boolean checkBool(MType argu, MType owner);
public boolean checkArray(MType argu, MType owner);
// 从owner中获取类argu，不存在则返回null
public MClass getClass(MType argu, MType owner);
// 从owner中获取方法argu，不存在则返回null
public MMethod getMethod(MType argu, MType owner);
```

3 从 MiniJava 到 Piglet

3.1 Piglet 简介

Piglet 是一种接近中间代码的语言，采用操作符在前的表达式，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，不是严格的三地址码。与 MiniJava 的区别在于：

- 取消了“类”的概念, 将其转换成一系列方法的组合
- 与抽象的数组相比，出现了存储的概念，有内存的申请和存取
- 用一系列 temp 变量代替了有变量名的变量

以下是 Piglet 语句的语义（常见的语句不作解释）:

- TEMP IntegerLiteral 临时单元
  TEMP 0, TEMP 1, ..., TEMP 19 用于传递调用参数，其它临时单元可以用作本过程内的局部变量 (且不用声明)
- NOOP
- ERROR
- PLUS Exp1 Exp2
- MINUS Exp1 Exp2
- TIMES Exp1 Exp2
- LT Exp1 Exp2
- JUMP Label
- CJUMP Exp Label
  如果 Exp 值为 1，执行下一条指令，否则跳转到 Label 处
- HALLOCATE Exp
  分配的内存大小需要是 4 的倍数
- MOVE TEMP * Exp
- HSTORE Exp1 IntegerLiteral Exp2
  将 Exp2 求得的值存储到地址 Exp1+IntegerLiteral 中
- HLOAD TEMP * Exp IntegerLiteral
  将地址 Exp+IntegerLiteral 中的值加载到临时单元 TEMP * 中

- CALL Exp1 ”(” (Exp#)* ”)”
  Exp1 为被调用过程的地址，各个 Exp# 作为调用参数
- Label ”[” IntegerLiteral ”]” StmtExp
  过程声明，Label 是一个过程起始地址，IntegerLiteral 是参数个数，StmtExp 是具体的过程内容
- ”BEGIN” StmtList ”RETURN” Exp ”END”
  复合嵌套语句，执行 StmtList 中的各语句，并返回 Exp 的值
- PRINT Exp

3.2 Piglet 语法

Goal	::=	MAIN StmtList ”END” (Procedure)* <EOF>
StmtList	::=	((Label)? Stmt)*
Procedure	::=	Label ”[” IntegerLiteral ”]” StmtExp
Stmt	::=	NoOpStmt
	|	ErrorStmt
	|	CJumpStmt
	|	JumpStmt
	|	HStoreStmt
	|	HLoadStmt
	|	MoveStmt
	|	PrintStmt
NoOpStmt	::=	NOOP
ErrorStmt	::=	ERROR
CJumpStmt	::=	CJUMP Exp Label
JumpStmt	::=	JUMP Label
HStoreStmt	::=	HSTORE Exp IntegerLiteral Exp
HLoadStmt	::=	HLOAD Temp Exp IntegerLiteral

MoveStmt	::=	MOVE Temp Exp
PrintStmt	::=	PRINT Exp
Exp	::=	StmtExp
	|	Call
	|	HAllocate
	|	BinOp
	|	Temp
	|	IntegerLiteral
	|	Label
StmtExp	::=	BEGIN StmtList ”RETURN” Exp ”END”
Call	::=	CALL Exp ”(” (Exp)* ”)”
HAllocate	::=	HALLOCATE Exp
BinOp	::=	Operator Exp Exp
Operator	::=	LT
	|	PLUS
	|	MINUS
	|	TIMES
Temp	::=	TEMP IntegerLiteral
IntegerLiteral	::=	<INTEGER_LITERAL>
Label	::=	<IDENTIFIER>

3.3 代码结构

以下代码位于 **minijava** 包中。

- minijava2piglet
  - Main.java 主程序
  - PigletOut.java 格式化地输出 Piglet 代码
    * indent 当前缩进长度

        * newLine 当前是否为新行

- symboltable 同上，略
- visitor
  - MakeSymbolTable.java 建立符号表 Visitor
  - TypeCheck.java 类型检查 Visitor
  - Minijava2Piglet.java 完成翻译的 Visitor
    * VTable 即 Variable Table，存储类的成员变量
    * DTable 即 Dispatch Table，存储类的成员方法
    * localVarTable 存储方法中局部变量到 Temp 临时单元的映射

3.4 实现步骤

步骤如下：

- 首先同上一步，建立符号表，检查类型错误
- 再通过一次 Visit 完成翻译过程，输出 Piglet 代码

3.5 数组的翻译

原来的创建数组、数组访问、获得数组长度的指令都已经不存在，将它们翻译为 Piglet 指令：

- **创建数组**
  使用 HALLOCATE 申请一片内存空间，在首地址处存储数组长度，在后面的空间分别存储数组每一个元素（可以将数组元素初始化为 0，避免 MiniJava 程序中由于数组元素未初始化引起空指针异常，但不是必须的），此空间首地址即作为数组的引用

- **数组访问**

  使用 HLOAD 获得数组元素；若数组未初始化，报 ERROR；若下标越界，报 ERROR

- **获得数组长度**

  使用 HLOAD 获得数组首地址存储的数组长度；若数组未初始化，报 ERROR

3.6 方法的翻译

由于不同类可能有同名方法，我们可以在方法名前加上类名，避免冲突。每一个方法翻译为 Piglet 中的一个 Procedure。方法需要知道其操作于哪一个对象实例，但 Java 中的 this 是作为一个隐式参数。翻译至 Piglet 我们可以约定 this 作为对应方法第一个参数 TEMP 0。

由于 Piglet 过程支持最多 20 个参数，若参数多于 20 个，需要将多余参数溢出至内存。此时可以约定将对应内存区域的引用储存在 TEMP 19 中。

我们可以在方法中将所有局部变量清 0，避免 MiniJava 程序中由于 TEMP 未初始化引起空指针异常，但不是必须的。

3.7 控制流的翻译

控制流的翻译比较容易，IfStatement 和 WhileStatement 容易翻译为跳转指令，不再赘述。值得一提的是，在对布尔表达式翻译时，编程语言通常都实现了短路求值的特性，对于包括 && 和 || 操作符的布尔表达式在进行求值时，只要最终的结果已经可以确定是真或假，求值过程便告终止。MiniJava 中包含 && 操作符，在第一个操作数为 false 时，不需要计算第二个操作数的值，这里的翻译可以通过条件跳转指令实现。

3.8 类的翻译

因为 Piglet 没有了“类”的概念，需要用较低层的指令实现类的功能。我们可以参考 C++ 面向对象中虚函数表的实现设计。我们从类的封装、继承、多态特性分别分析如何翻译类。

3.8.1 封装

类是对成员变量和成员方法的封装，所以需要申请两片内存空间，称为 VTable 和 DTable，分别存储成员变量的名称与成员方法的名称（表示方法的标签）。可以将 DTable 的首地址存储在 VTable 的首地址处，这样 VTable 即可以表示一个封装好的类的对象。我们可以在类初始化时将所有成员变量初始化为 0，避免 MiniJava 程序中由于成员变量未初始化引起空指针异常，但不是必须的。对类的对象引用时，若发现对象未初始化，报 ERROR。这里使用了几个函数来辅助类的翻译：

```
// 建立成员变量表、获取变量在表中的的offset
Vector<String> setVTable(MClass Class);
int getVTableOffset(MClass Class, String var);
// 建立成员方法表、获取方法在表中的的offset
Vector<String> setDTable(MClass Class);
int getDTableOffset(MClass Class, String var);
```

3.8.2 继承和多态

类的继承要求我们支持

- **子类继承父类的成员变量和方法**
  在构造子类的 Table 时，先将父类的 Table 内容按继承链自顶向下地拷贝到自己的 Table 中。这里的拷贝可以使用浅拷贝方法 clone()，此时 VTable 和 DTable 变量都只包含 String 成员，而不包含对公共资源的引用。翻译时再将 DTable 的地址添加在 VTable 中的首地址处。

- **子类增加新的变量、方法**
  可以将新增的变量、方法放置于父类的成员后面，作为子类的独有成员。

- **子类增加与父类同名的变量**（隐藏父类变量，不覆盖）
  在 Java 中将子类赋值给父类时成员变量会恢复父类的定义，即增加与父类同名的变量只会隐藏父类方法而不会覆盖。在上面的方法中，子类的成员变量位于父类

的成员变量后面，所以我们在子类中寻找成员变量时可以在 VTable 中从后向前查找，这样第一个找到的即为子类所定义的成员变量。

- **子类增加与父类同名的方法**（重写/覆盖父类方法）
  在 Java 中将子类赋值给父类时成员方法会保持子类的方法定义（多态），即子类会覆盖父类方法。所以，DTable 中子类重写的方法可以直接覆盖父类的方法。如上面所述，翻译时子类的 DTable 中的字符串为子类名 + 方法名，则可以自动覆盖父类的相应方法，调用子类的重写方法。

- **将子类赋值给父类**
  如上所述，Table 中先保存了继承链自顶向下的所有内容，才保存子类增加新的变量、方法。所以子类可以直接赋值给父类，父类根据其自己的 Table 大小访问此子类的 Table，那么就不会访问到子类独有的新的变量和方法。同时，子类重写的方法仍然以子类名为前缀，子类赋值给父类之后调用的仍是子类重写的方法，这样就实现了**多态**。

3.9 变量的翻译

变量可以分为以下几种：

- **类的成员变量**
  翻译在类的翻译中完成。访问时需要到对应的 VTable 中寻找，再 LOAD 到 TEMP 中。

- **方法的参数**
  翻译在方法的翻译中完成。访问时在 TEMP 0 到 TEMP 19 中寻找，若多余 20 个参数还需要在 TEMP 19 引用的内存空间中寻找，再 LOAD 到 TEMP 中。

- **方法内的局部变量**
  翻译在表达式的翻译中完成，被映射到一个 TEMP，映射关系存在 localVarTable 中。访问时在表中查找到对应的 TEMP，直接使用 TEMP。

4 从 Piglet 到 SPiglet

4.1 SPiglet 简介

Spiglet 与 Piglet 非常接近，主要区别包括:

- 没有 “嵌套表达式”，例如 Exp 不会嵌套 Exp，这更接近三地址代码
- 语句中，只有 move 可以使用表达式，print 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
- 表达式只有四种类型：简单表达式（临时变量、整数、标号），调用，内存分配，二元运算

4.2 SPiglet 语法

Goal	::=	MAIN StmtList ”END” (Procedure)* <EOF>
StmtList	::=	((Label)? Stmt)*
Procedure	::=	Label ”[” IntegerLiteral ”]” StmtExp
Stmt	::=	NoOpStmt
	|	ErrorStmt
	|	CJumpStmt
	|	JumpStmt
	|	HStoreStmt
	|	HLoadStmt
	|	MoveStmt
	|	PrintStmt
NoOpStmt	::=	NOOP
ErrorStmt	::=	ERROR
CJumpStmt	::=	CJUMP Temp Label
JumpStmt	::=	JUMP Label
HStoreStmt	::=	HSTORE Temp IntegerLiteral Temp

HLoadStmt	::=	HLOAD Temp Temp IntegerLiteral
MoveStmt	::=	MOVE Temp Exp
PrintStmt	::=	PRINT SimpleExp
Exp	::=	Call
	|	HAllocate
	|	BinOp
	|	SimpleExp
StmtExp	::=	BEGIN StmtList ”RETURN” SimpleExp ”END”
Call	::=	CALL SimpleExp ”(” (Temp)* ”)”
HAllocate	::=	HALLOCATE SimpleExp
BinOp	::=	Operator Temp SimpleExp
Operator	::=	LT
	|	PLUS
	|	MINUS
	|	TIMES
SimpleExp	::=	Temp
	|	IntegerLiteral
	|	Label
Temp	::=	TEMP IntegerLiteral
IntegerLiteral	::=	<INTEGER_LITERAL>
Label	::=	<IDENTIFIER>

4.3 代码结构

以下代码位于 **piglet** 包中。

- piglet2spiglet
  - Main.java 主程序
  - SpigletOut.java 格式化地输出 SPiglet 代码

    * indent 当前缩进长度
    * newLine 当前是否为新行

- visitor
  - GetMaxTempNo.java 获得 Piglet 代码中最大 Temp 编号的 Visitor，方便后续翻译中使用剩余的 Temp
  - Piglet2Spiglet.java 完成翻译的 Visitor

4.4 实现步骤

步骤如下：

- 第一次 Visit，获得 Piglet 代码中最大的 Temp 编号，方便后续翻译中使用剩余的 Temp
- 第二次 Visit，消除 Piglet 中的复合表达式，完成翻译

4.5 复合语句的消去

由于在 SPiglet 中只有 MOVE 可以使用 Exp 表达式，我们可以将 Piglet 中的所有 Exp 表达式递归地 MOVE 到 Temp 中，把 Exp 表达式出现的位置替换为对应的 Temp 即可。

5 从 SPiglet 到 Kanga

5.1 Kanga 简介

Kanga 是面向 MIPS 的语言，与 Spiglet 接近，但有如下不同:

- 标号（label）是全局的，而非局部的
- 几乎无限的临时单元变为了有限的寄存器: 24 个（a0-a3，v0-v1，s0-s7，t0–t9）

- 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，第一个值在 SPILLEDARG 0 中

- 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（注意: PASSARG 是从 1 开始的!）

- 过程的头部含 3 个整数 (例如: proc [5][3][4])
  参数个数、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）、过程体中调用其他过程的最大参数数目

5.2 Kanga 语法

Goal	::=	”MAIN” ”[” IntegerLiteral ”]” ”[” IntegerLiteral ”]” ”[” IntegerLiteral ”]” StmtList ”END” (Procedure)* <EOF>
StmtList	::=	((Label)? Stmt)*
Procedure	::=	Label ”[” IntegerLiteral ”]” ”[” IntegerLiteral ”]” ”[” IntegerLiteral ”]” StmtList ”END”
Stmt	::=	NoOpStmt
	|	ErrorStmt
	|	CJumpStmt
	|	JumpStmt
	|	HStoreStmt
	|	HLoadStmt
	|	MoveStmt
	|	PrintStmt
	|	ALoadStmt

	| AStoreStmt
	| PassArgStmt
	| CallStmt
NoOpStmt	::= "NOOP"
ErrorStmt	::= "ERROR"
CJumpStmt	::= "CJUMP" Reg Label
JumpStmt	::= "JUMP" Label
HStoreStmt	::= "HSTORE" Reg IntegerLiteral Reg
HLoadStmt	::= "HLOAD" Reg Reg IntegerLiteral
MoveStmt	::= "MOVE" Reg Exp
PrintStmt	::= "PRINT" SimpleExp
ALoadStmt	::= "ALOAD" Reg SpilledArg
AStoreStmt	::= "ASTORE" SpilledArg Reg
PassArgStmt	::= "PASSARG" IntegerLiteral Reg
CallStmt	::= "CALL" SimpleExp
Exp	::= HAllocate
	| BinOp
	| SimpleExp
HAllocate	::= "HALLOCATE" SimpleExp
BinOp	::= Operator Reg SimpleExp
Operator	::= "LT"
	| "PLUS"
	| "MINUS"
	| "TIMES"
SpilledArg	::= "SPILLEDARG" IntegerLiteral
SimpleExp	::= Reg
	| IntegerLiteral
	| Label
Reg	::= "a0"

| ”a1”
| ”a2”
| ”a3”
| ”t0”
| ”t1”
| ”t2”
| ”t3”
| ”t4”
| ”t5”
| ”t6”
| ”t7”
| ”s0”
| ”s1”
| ”s2”
| ”s3”
| ”s4”
| ”s5”
| ”s6”
| ”s7”
| ”t8”
| ”t9”
| ”v0”
| ”v1”

IntegerLiteral ::= <INTEGER_LITERAL>

Label ::= <IDENTIFIER>

5.3 代码结构

以下代码位于 **spiglet** 包中。

- spiglet2kanga
  - FlowGraph.java 流图类
    * vVertex 流图节点数组
    * mVertex 节点编号 vid 到流图节点的映射
    * callPos 调用了其他方法的节点集
  - FlowGraphVertex.java 流图节点（基本块）类
    * vid 节点编号
    * Pred 前驱节点集
    * Succ 后继节点集
    * Def 在此基本块中被定义的节点编号集
    * Use 在此基本块中被使用的节点编号集
    * In 进入此基本块时活跃的节点编号集
    * Out 离开此基本块时活跃的节点编号集
  - LiveInterval.java 活性区间类
    * begin 区间起始点
    * end 区间结束点
    * tempNo 此区间对应的 TEMP 编号
  - Method.java 方法类
    * methodName 方法名
    * paramNum, stackNum, callParamNum 即 Kanga 中方法头部的三个整数
    * regT, regS, regSpilled 分别记录 T 寄存器，S 寄存器，SPILLEDARG 的分配情况
    * mTemp 从 TEMP 编号到对应活性区间的映射
    * flowGraph 此方法的流图

  - Temp2Reg.java 完成从 TEMP 到 Reg 的寄存器分配
    * LiveAnalyze() 活性分析
    * GetLiveInterval() 获取活性区间
    * LinearScan() 线性扫描分配寄存器
  - Main.java 主程序
- visitor
  - GetFlowGraphVertex.java 获取流图节点（基本块）的 Visitor
  - GetFlowGraph.java 获取流图的 Visitor
  - Spiglet2Kanga.java 完成翻译的 Visitor

5.4 实现步骤

步骤如下：

- 第一次 Visit，获取基本块
- 第二次 Visit，建立流图
- 活性分析，得到每一个 TEMP 的活性区间
- 寄存器分配，这里使用线性扫描算法
- 第三次 Visit，完成翻译

5.5 流图的建立

为了简便起见，将一个语句作为一个基本块。第一次 Visit 过程中，可以为每一个语句编号，作为基本块的编号。同时，可以记录程序中的所有方法和 Label，便于后续处理；为每一个 TEMP 初始化一个对应的活性区间对象（取 TEMP 第一次出现的位置为区间起点，若 TEMP 为方法的参数则区间起点设为 0；区间终点在活性分析中获得）；

记录方法中调用其他方法的最大参数个数 callParamNum，这是 Kanga 中方法头部所需的信息之一。

然后就可以建立流图。在图中加一条边 (i, j) 时，将 j 加入 i 的后继点集 Succ，将 i 加入 j 的前驱点集 Pred。相邻的语句之间需要连边；除此之外，跳转指令、调用指令也需要连上对应的边。

5.6 活性分析与活性区间

在流图建立完成后，我们可以进行活性分析。对于两个相连接的基本块的中间时刻，如果此时刻的一个变量已经被 define，在未来还需要被 use（且该 use 之前不重新发生 define），则称该变量此时是活跃的。在活性分析中，我们可以从后向前迭代计算每个基本块入口 In 和出口 Out 处的活跃节点集，当迭代到达不动点时即得到了流图中全部位置的活跃点集。算法的过程如下：

Algorithm 1: Liveness Analysis

Input:

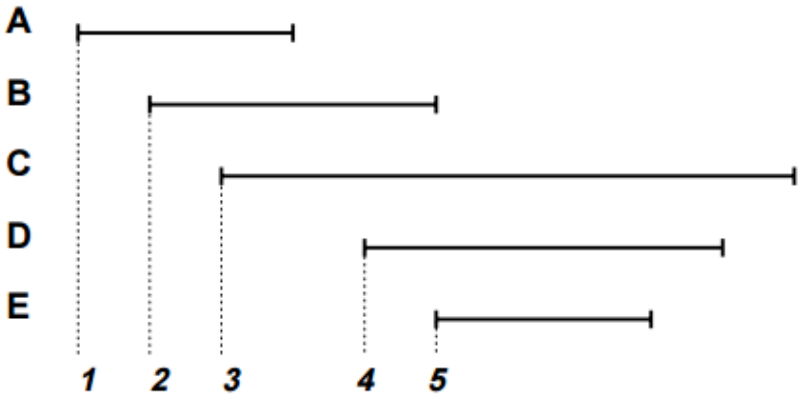
- $N \equiv$ basic block node set
- $Exit \equiv$ last node in basic blocks

Data:

- $IN \equiv$ set of variables live at start of block
- $OUT \equiv$ set of variables live at end of block
- $USE \equiv$ set of variables with upwards exposed uses in block (use prior to definition)
- $DEF \equiv$ set of variables defined in block prior to use

```
begin
    for n ∈ N − {Exit} do IN[n] ⟵ ∅;
    OUT[Exit] ⟵ ∅
    IN[Exit] ⟵ USE[Exit]
    Changed ⟵ N − {Exit}
    while Changed ≠ ∅ do
        choose a node n ∈ Changed
        Changed ⟵ Changed − {n}
        OUT[n] ⟵ ∅
        for s ∈ successors(n) do
            OUT[n] ⟵ OUT[n] ∪ IN[s]
        IN[n] ⟵ USE[n] ∪ (OUT[n] − DEF[n])
        if IN[n] changed then
            for p ∈ predecessors(n) do
                Changed ⟵ Changed ∪ {p}
end
```

得到流图中全部位置的活跃点集之后，我们就可以更新每一个变量对应的活性区间的终点（最后活跃的位置），得到所有变量的活性区间。以下是活性区间的一个示例图：



5.7 寄存器分配算法

5.7.1 图着色算法

图着色算法的思路如下。对所有 TEMP 建立干涉图：图中的每一个节点表示一个 TEMP，若两个 TEMP 在某时刻互相干扰，则在两节点之间连一条边；然后采用图论中的图着色算法对干涉图 G 进行着色，色数为寄存器个数，如果两个顶点之间存在一条边，则它们不能使用相同的颜色。利用图着色的算法给干涉图着色之后，就可以给相同颜色的顶点分配同一个寄存器。若存在不能染色的节点，则需要溢出到内存。图着色算法的时间复杂度为 $O(n^4)$。这里我们采用另一种算法，即线性扫描算法来实现寄存器分配。

5.7.2 线性扫描算法

线性扫描算法的时间复杂度为 $O(V \log R)$（V 个变量、R 个寄存器）。线性扫描算法的步骤如下：先将所有活性区间按照起始点排序（起始点相同的可按结束点排序），依次为这些区间分配寄存器；每次先检查这个区间之前的区间是否已经结束，若结束则将它对应的寄存器释放；然后在当前空闲的寄存器中合理地选择一个分配给当前的活性区间；若当前没有空闲寄存器，则需要寄存器溢出：将活跃区间中结束最晚的一个对应的变量值溢出，将该寄存器分配给当前区间。算法过程如下图所示。

在寄存器分配过程中，S 寄存器为调用者保存，而 T 寄存器为被调用者保存。作为一种优化，寄存器分配时对于一个 TEMP，如果它的活性区间跨调用，则分配 S 寄存器或溢出；如果它的活性区间不跨调用，优先分配 T 寄存器，其次 S 寄存器，最后溢出。寄存器分配结束之后，就可以知道当前方法需要多少栈空间（包括多于 4 个的参数，溢出的 TEMP，保存的寄存器），从而计算出 Kanga 所需的方法头部信息 stackNum。

```
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
```

5.8 参考文献

线性扫描算法：Massimiliano Poletto and Vivek Sarkar, Linear Scan Register Allocation, ACM TOPLAS 21(5):895-913, 1999.

http://web.cs.ucla.edu/~palsberg/course/cs132/linearscan.pdf

6 从 Kanga 到 MIPS

6.1 MIPS 简介

MIPS（Microprocessor without Interlocked Pipeline Stages）是一种采取精简指令集（RISC）的处理器架构。我们使用 SPIM 模拟器运行 MIPS 代码，运行我们编译得到的 MIPS 代码并与 Java 的运行结果相比较。可以使用 GUI 版本观察寄存器状态，也可以直接通过命令行的形式运行，例如：

```
spim -file Factorial.s
```

6.2 MIPS 语法

MIPS 的语法是熟知的。这里列出我们需要用到的语法规则：

6.2.1 汇编信息

- .text 代码段 (指令段)
- .data 数据段
- .globl 全局符号声明
- .align 标记对齐
- .asciiz 字符串 (带终止符)

6.2.2 寄存器约定

- a0 - a3: 存放向子函数传递的参数
- v0 - v1: 存放子函数调用返回结果，还可用于表达式求值
- 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 - t8: 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

- sp: 栈 (stack) 指针
- fp: 帧 (frame) 指针
- ra: 返回地址 (用于过程调用)

6.2.3 MIPS 指令

- 运算

  addu, add, subu, sub, mul, and, neg

- 常数操作

  li

- 数据传输

  la, lw, sw, move

- 比较

  seq, slt

- 控制指令

  b label: 分支到 label

  beqz Rsrc, label: 如果 Rsrc 为 0 就分支到 label

  bgeu Rsrc1, Src2, label: 如果 Rsrc1 大于等于 Src2 就分支到 label

  j label: 无条件转移到 label

  jal label:(jump and link) 无条件转移到 label，并将下一指令的地址写入 $ra

  jalr Rsrc: 使用寄存器的跳转指令，指令的跳转地址在 Rsrc 寄存器中，并将下一指令的地址写入 $ra

- 系统调用指令

  syscall

  $v0 中包含调用号 (共 12 个):

  1: 打印整数, 数字在 $a0 中

  4: 打印字符串, 字符串在 $a0 中

9: 申请内存块, 申请长度在 $a0 中
所获得内存的地址在 $v0 中

6.2.4 运行栈

运行栈结构从高地址到低地址如下:

上一帧	0($fp)
$ra $fp 溢出的 TEMP 调用者保存的 S 寄存器 参数 X ... 参数 4	0($sp)
下一帧	

6.3 代码结构

以下代码位于 **kanga** 包中。

- kanga2mips
  - Main.java 主程序
  - MIPSOut.java 格式化地输出 MIPS 代码
- visitor
  - Kanga2MIPS.java 完成翻译的 Visitor

6.4 实现步骤

步骤如下:

- 一次 Visit，将 Kanga 翻译为 MIPS 目标代码

6.5 运行栈的维护

Kanga 到 MIPS 的翻译比较容易，这一阶段的主要内容在于运行栈的维护。运行栈的格式如上所述。每个方法首尾部分都需要加上一些维护栈的指令，主要在于计算栈中不同内容所在位置的偏移量，例如对 SpilledArg、PassArgStmt、保存的寄存器等内容，应该正确的计算出对应的栈位置。MIPS 加入的汇编信息、代码末尾的几个系统调用函数是较为模式化的，不需要做太多改动。对于 MOVE Reg Exp 类型的 Kanga 指令，由于 Exp 有各种不同的类型，BinOP 也有不同类型的版本，所以精确翻译比较繁琐；对于立即数等也可以直接先加载到寄存器再计算，可以减少繁琐程度，相应地性能有所减弱。

7 总结

7.1 各模块的整合

以下代码位于根目录下。

- Main.java 整合上述的五个阶段，生成从 MiniJava 到 MIPS 的 end-to-end 的编译器。

对于五个模块的整合，我采用了 java.io 中的 PrintStream、ByteArrayInputStream、ByteArrayOutputStream 等类，对每一阶段的输入与输出做了重定向，使 Main 类接受标准输入流中的 MiniJava 代码输入，经过上述分阶段的处理，前一个阶段的输出字符串作为下一个阶段的输入，最终生成 MIPS 目标代码至标准输出流。

7.2 测试结果

使用在课程网站上给出的 8 个样例程序进行测试。它们在以上各个阶段均得到了正确的结果。

以下选择了两个比较复杂的样例，简单测试了一下编译器的效果。可以看到，我们的编译器速度还不错，但生成代码冗余较多、目标文件较大。编译过程中编译优化是无止境的，我的实现中优化相对较少。

	BinaryTree	TreeVisitor
Javac 编译时间	real,0m0.693s user,0m1.380s sys,0m0.052s	real,0m0.704s user,0m1.612s sys,0m0.052s
Java class 运行时间	real,0m0.098s user,0m0.096s sys,0m0.012s	real,0m0.063s user,0m0.056s sys,0m0.012s
Java class 总文件大小	4.2KB	5.8KB
MiniJava2MIPS 编译时间	real,0m0.546s user,0m1.368s sys,0m0.060s	real,0m0.548s user,0m1.488s sys,0m0.072s
MIPS 代码 运行时间	real,0m0.023s user,0m0.016s sys,0m0.000s	real,0m0.013s user,0m0.008s sys,0m0.004s
MIPS 代码 文件大小	39.1KB	45.7KB

7.3 项目重点

我认为，本次编译器设计的重点有以下几部分：

- 理解 Visitor 设计模式与抽象语法树
- 设计完备且便利的符号表
- 全面的类型检查
- 类的底层实现
- 活性分析与寄存器分配
- 代码优化

7.4 课程收获

本课程的项目比较具有挑战性，编写编译器是一项复杂的工程，即使在已有词法分析、语法分析工具的基础上，依然有比较多的难点与细节需要注意，工程量比较大。当然，完成项目过程中的收获也是巨大的。编译实习让我更加深刻地了解了编译原理中讲授的内容。将编译器分阶段设计可以让代码更加层次化，便于将不同的需求分层实现，并且更容易调试错误。编译器这种良好的系统架构对于设计其他大型软件/硬件/系统都具有指导意义。

在调试的过程中，最常出现的问题就是空指针异常，往往是因为某个地方逻辑考虑不全造成了访问空指针的现象。这个项目锻炼了我设计复杂逻辑的能力，编译器是一个逻辑严密的工具，必须每一个阶段都经过良好的设计才能得到正确的结果。

编译器设计中的优化也是具有挑战性的。编译优化需要在保证正确的前提下完成提高运行速度/提高编译速度/减少空间占用/减少冗余代码等工作，相应地维护难度也大大上升。编译优化同时也与体系结构中的缓存、流水线等内容相关，是比较复杂的。当然，编译优化对于程序设计者来说也是具有重要意义的。

7.5 课程建议

本课程中从 MiniJava 到 MIPS 的工具链是非常完备的，每一步都可以方便地进行阶段性测试。在 JavaCC 和 JTB 的帮助下，我们可以方便地遍历抽象语法树，相应地对词法分析、语法分析的过程就比较缺少实践，也让实习和理论课的进度相差较大。所以还是建议在编译原理课程后再选修编译实习，或者调整两门课的进度让两门课配合的更好。

编译实习的五个步骤中，第三步和第五步的工作量与其他步骤相比较轻松，建议可以和第二步、第四步分别合并。目前每个阶段两周的作业期限还是感觉比较紧张，最好能够同时延长作业期限。除此之外，建议课程能够有更加明确的评分标准，也许能更好地激励大家在编译各阶段内在基础功能上做更多的优化工作等。

这门课虽然只是 2 学分的课程，但这学期我为它付出了不少时间，也感到收获很大。老师的讲解、课件都很有帮助，助教提供的自动评测服务器非常实用，这学期内助教一个人面测全班的同学也十分辛苦…总之，感谢老师和助教一学期的辛勤付出！