编译实习实习报告

北京大学 18 春

2018-6-27

作者 1：蒋钰钊 1500012945
作者 2：易宇轩 1500012945

目录

0 简单说明

本项目是 minijava 到 MIPS 的编译器，由 1500012945 蒋钰钊和 1500012811 易宇轩实现。

下面进行简要说明：
在 MiniJava（注意是大写，这个是总文件夹，里面有 kanga, lib, minijava, piglet, spiglet, test 文件夹）中的 Main.java 文件是集合性文件。在这个文件中有一行
InputStream in = new FileInputStream("test/Factorial.java");
把 Factorial 换成 test 文件夹下的任意一个非 error 后缀的 java 文件，运行 Main 函数即可。

```
InputStream in = new FileInputStream("test/Factorial.java");
//PrintStream out = new PrintStream("MoreThan4.asm");
//InputStream in = System.in;
// PrintStream out = System.out;
PrintStream out = new PrintStream("mine.s");
```

之后刻在 Minijava 文件夹下得到 mine.s，这个就是 MIPS 文件，可以用专门的软件进行测试。

1 编译器概述

1.1 基本功能

本编译器以 minijava 的源代码为输入，输出为 MIPS 汇编代码。编译过程分为类型检查，中间代码生成，中间代码简化，寄存器分配，MIPS 代码生成五步。

1.2 实现步骤

先遍历一遍语法树，构造符号表，完成简单信息的填写，包括类继承关系，方法和变量的从属关系等。

之后再遍历一遍语法树进行类型检查。

1.3 编译器特点

Minijava 是 Java 的子集，其语法规则与 java 相比简单很多。给本编译器的实现带来很多便利。涉及到编译器实现的具体数据结构与算法将在下文正文具体实现部分给出，在此列举 minijava 对 java 语法的几处化简（包含一些探索出的化简）：

1、不允许方法重载

2、类中只允许申明变量和方法,不允许出现嵌套类

3、只有类,没有接口

4、表达式类型共有 9 种:加,减,乘,与,小于,数组定位,数组长度,消息传递(即参数传递),基本表达式

5、基本表达式共有 9 种:整数(Integer),”真”(true),”假”(false),对象,this,初始化(allocation),数组初始化(array allocation),非(not),括号(bracket)

6、数组只有 int 型数组
7、数组访问表达式中不允许出现符号，如 a[3+3]

2 编译器设计

2.1 Minijava 语法

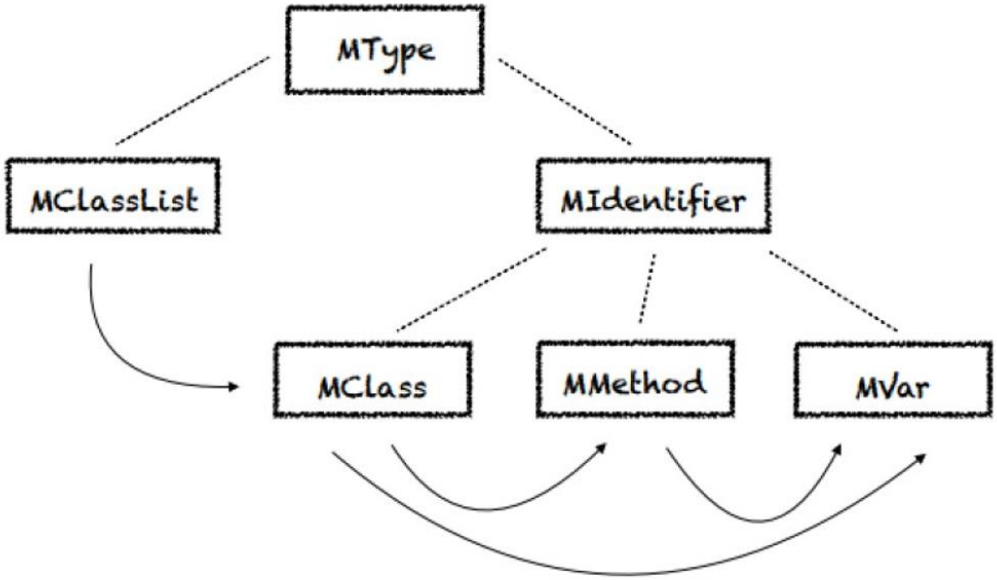
```
Goal                    ::= MainClass ( TypeDeclaration )* <EOF>
MainClass               ::= class Identifier ”{” ”public” ”static” ”void” ”main” ”(”
                            ” String” ”[” ”]” Identifier ”)” ”{” ( VarDeclaration )*
                          ( Statement
                            )* ”}” ”}”
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 代码设计

首先是数据结构方面的设计，设计出的数据结构关系如图。
其中 MType 是基本类，里面的基本变量是 String 类变量表示这个类型名，以及出现的 line 和 column

MClassList 是代表类集合的数据结构，采用单例模式，里面存储了所有出现的 MClass。
MClass 是代表类的数据结构，里面存储了类名和类内方法 MMethod，以及类内变量 MVar。
MMethod 代表类内方法，需要存储用到的参数，参数列表（方法签名）以及临时变量 MVar。
MVar 代表变量，有着变量自己的类型名（或变量名，两用）



虚线表示继承关系、实现表示包含关系
此图来自课件

另外由于函数调用需要检查形参和实参的匹配情况，因此使用了 MCallList 结构来记录实际传递参数的列表。
这个图来自课件，我们再设计的时候实际上没有 MIdentifier 这个类，相当于 MClass 等直接继承自 MType！

2.3 类型检查

2.1.1 目标

关于是否代码是否符合词法规范与语法规范的问题，由 minijava 定义的产生式与 javacc 与 JTB 生成的语法树已经可以解决，不符合语法规范的程序会在 minijavaparse.java 处直接抛出异常，因此类型检查只需要检查代码是否符合语义规范。

需要检查的错误有：

- 1. 使用未定义的类、变量和方法
- 2. 重复定义类、变量和方法
- 3. 类型不匹配
    - if 、while 的判断表达式必须是 boolean 型
    - Print 参数必须为整型
    - 数组下标必须是 int 型

    - 赋值表达式左右操作数类型匹配
- 4. 参数不匹配
    - 类型、 个数、return 语句返回类型
    - 不允许重载
- 5. 操作数相关：+、*、< 等操作数须为整数
- 6. 类的循环继承 、多重类的循环继承
- 7. 数组越界
- 8. 使用未初始化的变量
- 9. 7 和 8 外的其他 Bonus …

2.1.2 代码说明

本阶段包含的代码有

- 包 minijava.typecheck
- 包 minijava.symboltable
- 类 minijava.visitor.SymbolTableVisitor
- 类 minijava.visitor.TypecheckVisitor

2.1.3 方法概述

本阶段的主要操作是分别调用 SymbolTableVisitor 和 TypeCheckVisitor 两个 GJVisitor 对语法树进行两次遍历.具体如下：

1、 第一次遍历语法树，在全局符号表中填入类的信息，在类中填入函数,变量以及相应信息，在函数中填入变量信息。在添加类，方法，和变量式，检查同一作用域中是否存在重复定义的问题；

2、 第二次遍历语法树,检查其余错误,具体检查时机：

(a)在定义一个新对象时,检查类型是否已声明，在使用函数/变量时，检查当前环境下是否有该方法/变量；

(b)在赋值节点,检查类型是否匹配；

(c)在消息传递节点（这实际上是函数调用节点）,检查参数个数,类型是否匹配；

(d)在定义一个新的继承类时,检查是否存在该类的定义，同时检查是否有循环继承。

(e)在变量赋值节点，若右值没有使用未定义的表达式，同时是确定值（右值是仅由字面量构成的算数表达式或逻辑表达式），则把左值变量标记为已初始化且初始化为定值，否则就标记为初始化但是未定值。

(f)对每个表达式检查是否有使用未初始化变量。

(g)对每个数组访问/赋值表达式，检查数组是否初始化，若初始化且长度确定，那么检查数组访问指针表达式，若指针访问表达式没有使用未初始化变量且是确定值（仅由字面量构成的算数表达式）那么检查数组访问是否越界；对数组访问指针不是确定值，如通过函数调用得到返回值再访问的情况，不能判断是否越界。

3、在上述过程中，每次遇到问题便打印具体错误。

核心代码和具体注释如下：

// 构成记录类内信息，方法信息，变量信息的基础结构，看变量是否初始化，数组是否有长度等功能都集成在里面

```
public class MType {
	// ---for typecheck, bonus part begins---
	protected int hasInitLength = 0;
	// 0: has initialized, but not sure of value
	// -1: has not initialized
	// 1: has initialized, and sure of value
	protected int length;
	public int getHasInitLength() {
		return hasInitLength;
	}
	public void setHasInitLength(int hasInitLength) {
		this.hasInitLength = hasInitLength;
	}
	public int getLength() {
		return length;
	}
	public void setLength(int length) {
		this.length = length;
	}

	protected int hasInit = 0;
	// 0: has initialized, but not sure of value
	// -1: has not initialized
	// 1: has initialized, and sure of value
	protected int intValue;
	protected boolean booleanValue;
	public int getHasInit() {
		return hasInit;
	}
	public void setHasInit(int hasInit) {
		this.hasInit = hasInit;
	}
	public void setIntValue(int intValue) {
		this.intValue = intValue;
	}
	public void setBooleanValue(boolean booleanValue) {
		this.booleanValue = booleanValue;
	}
	public int getIntValue() {
		return intValue;
	}
	public boolean getBooleanValue() {
		return booleanValue;
	}
```

```
        // 重写 equals 和 hashCode 使得可以在加入 HashSet 时保证不重复，equals
    方法在 minijava2piglet 阶段有更改！
        @Override
        public boolean equals(Object obj) {
            if (!(obj instanceof MType)) {
                return false;
            } else {
                return name.equals(((MType)obj).name);
            }
        }

        @Override
        public int hashCode() {
            return name.hashCode();
        }
}

// 构成变量的结构
public class MVar extends MType {
        protected String type;
        protected String methodName;
        protected String className;
}

// 记录方法名，方法返回值，参数列表和变量列表
public class MMethod extends MType {
        protected String returnType;
        protected String className;
        protected ArrayList<MVar> mParamArrayList = new ArrayList<MVar>();
        protected HashSet<MVar> mVarSet = new HashSet<MVar>();
}

// 记录类内信息，包括类名，方法名和变量名等
public class MClass extends MType {
        protected HashSet<MMethod> mMethodSet = new HashSet<MMethod>();
        protected HashSet<MVar> mVarSet = new HashSet<MVar>();

        protected String parentClass;
}

// 该结构是全局类结构，负责记录全局里的类情况。
public class MClassList extends MType {
        public static MClassList instance = new MClassList();
```

```
    private static int tempNumber = 20,labelNumber = 0;

    private HashSet<MClass> mClassSet = new HashSet<MClass>();
}

// 该结构用来记录函数调用的实参情况，以及属于哪个函数，与函数运行上下文
public class MCallList extends MType {
    protected MMethod callerMethod;
    protected MMethod contextMethod;
    protected ArrayList<MVar> mVarArrayList;
}
```

以上结构的“记录”，都是有带有实际 Set 或者 Array 的，可以通过遍历从中得到需要的数据。

2.1.4 SymbolTableVisitor

在建立符号表的 SymbolTableVisitor 中，主要工作是识别类定义和填入类信息。建立符号表就需要的信息有：

全局类符号表：

• 每一个类

每一个类名，继承关系，类内包含的变量，类内包含的函数，函数的参数列表，和函数内部的变量。其中通过第二个参数表示当前上下文。

此时可以做基本检查，检查类，方法，和变量定义是否重复。

2.1.5 TypeCheckVisitor

进行类型检查的时候基于建立好的符号表，基本思想如下：

1） 碰到类继承的时候检查类是否有定义，是否出现循环继承

2） 碰到变量声明的时候检查变量类型是否存在

3） 碰到变量使用的时候检查变量是否存在

4） 碰到变量使用的时候检查变量类型是否符合表达式要求的类型，表达式结点会返回表达式运算后的类型（其实和变量类型相同）

5） 碰到方法调用的时候检查方法是否有定义，检查方法调用形参和实参是否匹配

6） 检查各种其他的类型不匹配，比如 if 和 while 内的表达式一定要是 boolean 类型等

Bonus 部分其实只能算作伪 bonus，因为这些错误都不能在编译时被检查，发现问题之后把报错改为警告，只要程序员确认代码没有问题，依然可以运行。

Bonus 实现了初始化检查，即静态判断一个变量使用的时候是否已经被初始化，数组是否已经分配空间，以及是否越界。但是这实际上是不可能判断的。以一个例子来佐证：

funA 中使用数组 a，funB 中初始化数组 a，长度为 10，funC 中初始化数组 a，长度为 20.现考虑 funA 中访问 a 的第 15 个元素。静态检查就需要检查距离 a 的使用最近的函数调用，是调用了 funB 还是 funC 才能确定 a 是否初始化以及是否越界。而这样的判断代价不小。当出现跨多个文件多个类的联合编译时情况尤其复杂，不如在运行时交由虚拟机检查，是否访问了非法地址。

整个类型检查阶段，用到了如下辅助函数：

```
/**
  * check whether a class has been defined
  * @param name:name of the class to be checked
  * @param line:begin line of the object of this class
  * @param column:begin column of the object of this class
*/
public boolean classCheck(String name,int line,int column)

/**
  * check whether a method has been defined
  * @param name: method name
  * @param nClass: in which class
  * @param line
  * @param column
  */
public boolean methodCheck(String name, MClass nClass, int line, int column)

/**
  * check whether a var has been defined
  * @param name:name of the target var
  * @param argu:a method or a class
  * @param line:begin line of the var
  * @param column:begin column of the var
  */
public boolean varCheck(String name,Object argu,int line,int column)

/**
  * check whether a expression is the given type
  * @param exp:a expression
  * @param type:the expression is supposed to be this type
  * @param printContent:content to be printed if not match
  */
public boolean checkExpType(MType exp,String type,String printContent)

/**
* @param nType: the type is from identifier, the name is var name
* @param argu: which field the var is from
* @return the type of the var
*/
public String changeIdentifier(MType nType,Object argu)

/**
* for every expression, we need to chech whether it has been initialized.
* @param exp: expression
```

```
     * @param printContent: if the analysis goes wrong, what content we should
print
     * @return if expression has not been initialized, return true
     */
     public boolean checkExpNotInit(MType exp, String printContent)

     /**
     * @param exp: expression
     * @return if the expression has a definite value(either integer or boolean),
return true.
     */
     public boolean checkExpDefinite(MType exp)

     /**
     * @param exp:expression in the method should be an integer array indentifier.
     * @return if the identifier is an integer array and the array has a definite length,
return true.
     */
     public boolean checkExpDefiniteLength(MType exp)

     /**
     * to check whether a visit to an array is valid
     * @param exp1: exp1 should be an integer array indentifier.
     * @param exp2: exp2 shoulb be an integer number
     */
     public void checkOutOfRange(MType exp1, MType exp2)
```

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

• symboltable 同上，略

• visitor

- SymbolTableVisitor.java 建立符号表 Visitor
- TypeCheckVisitor.java 类型检查 Visitor
- Minijava2PigletVisitor.java 完成翻译的 Visitor

其中 SymbolTableVisitor 和 TypeCheckVisitor 的实现和类型检查部分类似。

3.4 实现步骤

主要步骤如下：

• 首先同上一步，建立符号表，检查类型错误

• 其次对类进行补完和处理

• 再通过一次 Visit 完成翻译过程，输出 Piglet 代码

核心代码如下：

```
MClassList.instance.completeClass();
MClassList.instance.allocTemp(20);
```

这两个函数完成了类的补完和翻译 Piglet 代码前的预处理工作

翻译的核心思想：只有方法需要翻译，一次主程序的执行只是调用了一群方法，方法访问了类内变量和局部变量，但是这些都不需要翻译，只需要绑定在方法内部即可。多态的翻译思想也由此而生：只要知道访问哪个函数即可，函数内部访问的具体变量实际都是写好的。翻译的核心就是翻译方法。

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，此空间首地址即作为数组的引用。

• 数组长度访问

使用 HLOAD 获得数组首地址存储的数组长度

• 数组访问

首先得到数组引用的地址，之后根据访问的元素计算出实际访问的元素地址偏移，再使用 HLOAD 获得需要访问的数组元素

3.6 方法的翻译

• 方法声明

由于在 allocTemp 中已经对方法内的变量进行了初始分配，因此方法声明部分的翻译只要把语句翻译好就行

• 方法调用

方法调用的核心是 getmClass，该语句负责得到运行时类型。该语句把类的运行时判断在 Expression 以下抹平。仅由非基本类会被记录 class，对于基本类的几个语句简单记录为缺省值即可，因为它们并不涉及多态的问题。在 Expression 这一层次，所有的表达式的返回值只要不是基本变量都会被记录为运行时类。

```
/**
 * f0 -> AndExpression()
 *       | CompareExpression()
 *       | PlusExpression()
 *       | MinusExpression()
 *       | TimesExpression()
 *       | ArrayLookup()
 *       | ArrayLength()
 *       | MessageSend()
 *       | PrimaryExpression()
 */
public MPiglet visit(Expression n, Object argu) {
    MPiglet _ret = n.f0.accept(this, argu);
    return _ret;
}
```

而实际进行方法调用的时候，找到运行时类，并且由于 completeClass 做过了，可以找到对应方法在方法表中的 offset，直接取出访问即可。这也依赖于对对象的赋值操作是引用的赋值（即变量表首地址的赋值，变量表和运行时类是对应的）

```
/**
 * f0 -> PrimaryExpression()
 * f1 -> "."
 * f2 -> Identifier()
 * f3 -> "("
 * f4 -> ( ExpressionList() )?
 * f5 -> ")"
 */
public MPiglet visit(MessageSend n, Object argu) {
    MPiglet _ret = new MPiglet("CALL");
    _ret.appendCode(new MPiglet("BEGIN"));
    MPiglet _f0 = n.f0.accept(this, argu);
    MClass mClass = _f0.getmClass();
```

控制流的翻译

CJUMP 的功能是在表达式为 false 的时候跳转到固定标号。这部分翻译相对容易，只要画出控制流图很容易明白。

类的翻译

通过 completeClass 和 allocTemp 这两个核心函数完成翻译。对继承的变量简单地自顶向下增加即可。对于继承的方法判断是否覆盖，若是覆盖则不用继承该方法。最后对本类中尚未

进行 offset 分配的变量和方法进行 offset 分配，注意变量的 offset 不能乱，但是方法无所谓，只要每一个类自己统一即可。

• 对象声明

构造变量表和方法表，按照规定的 offset 填入方法，并把类内变量全部初始化为 0（或 null），最后返回变量表首地址

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

包含了代码转换的主函数

• visitor

包含了 Piglet2SpigletVisitor.java 这个 visitor

4.4 代码逻辑

首先通过正则表达式找到当前 piglet 代码中的最大 temp 值，以后要再用到新的临时 TEMPNUM 时，就从这里开始计数，防止 temp 值重用混用出错。

之后用 Piglet2SpigletVisitor 依次访问节点。

因为 splglet 的语法很严格，在 piglet 里面可以用 Exp 的地方，在 spiglet 中大多数都只能用 TEMP。

所以遇到任何 Exp，都先_ret.appendCode("MOVE TEMPNUM Exp")，把它保存到一个新的临时 TEMP 里面，并且记录下这个 TEMPNUM；以后需要用到这个代码段的时候，直接用这个 TEMPNUM 就行了。

不过，还是有一部分情况，可以直接用 Exp 或 SimpleExp，不必用 TEMP。如果所有地方都写成 TEMP，未免太过冗余。所以遇到任何 Exp，除了把它保存到一个新的临时 TEMP 里面，也保存一份它的 Exp 版本。如果这个 Exp 还属于 SimpleExp，还保存一份它的 SimpleExp 版本。

1）如果在需要用到这个代码段的时候，可以直接用 Exp 或 SimpleExp 的版本，那就直接用，不需要_ret.appendCode("MOVE TEMPNUM Exp")了。同时把这个 TEMPNUM 释放、回收重用。

2）如果在需要用到这个代码段的时候，发现它就是类似 L2 的 Label 而不是函数名 label、或者它是字面值：那么更是可以直接回收它的 TEMPNUM 了。

4.5 需要注意的点

（1）SPiglet 中 begin end 只能在函数调用时出现，尽管 StmtExp 产生式中有"BEGIN"，"RETURN"，"END"，并且作为 Exp 产生式的一种（这个相当于把变量表的分配和 return 作为一个完整语句），但是不能在这里依序翻译"BEGIN"，"RETURN"，"END"，只能在方法的产生式中翻译这些部分。对于这个语句的 RETURN 值，记录下它返回值被存在哪个寄存器即可。

（2）当 exp 中涉及到其他的 TEMP 时，要小心代码添加的顺序。例如 piglet 里可以写：Operator Exp1 Exp2，在 spiglet 里只能写成：Operator TEMP SimpleExp

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。

- symboltable
  - FlowGraph.java 流图类
    * 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

-Context.java 上下文，进行代码生成的包装

-FlowGraphPointer.java 流图指针，用来建立流图，是一个全局信息

-FlowGraphVertex.java 流图节点，指示每一句语句的前后句

-LiveInterval.java 活跃区间，维护每一个 TEMP 的活跃区间

-Method.java 维护每一个方法的重要信息

-RegAssignment.java 实际进行寄存器分配

- visitor

-CreateFlowGraphVertexVisitor.java 创建流图节点

-CreateFlowGraphVisitor.java 创建流图，把节点拓扑相连

-Spiglet2KangaVisitor.java 代码翻译

5.4 实现步骤

步骤如下：

- 第一次 Visit，把每一句语句作为一个节点，初始化节点序号，并记录每一个 TEMP *第一次出现的语句序号，为活跃变量分析做准备
- 第二次 Visit，在第一次 Visit 的基础上，把每个语句节点的按照前后语句顺序进行拓扑连接，为 def，use 的活跃变量分析做准备
- 进行活跃变量分析，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，完整寄存器分配
- 第三次 Visit，利用寄存器分配时的寄存器分配信息和栈信息完成翻译

5.6 活跃变量分析和寄存器分配

实现核心是活跃区间，通过以下数据结构实现：

```
public class LiveInterval implements Comparable<LiveInterval> {
    public int begin, end;
    // S or T
    public boolean S = false;
    public int tempNo;

    public LiveInterval(int tempNo, int begin, int end) {
        this.begin = begin;
        this.end = end;
        this.tempNo = tempNo;
    }

    public int compareTo(LiveInterval another) {
        // compare Interval [begin, end]
        if (begin == another.begin)
            return end - another.end;
        else
            return begin - another.begin;
    }
}
```

该结构中的 S 表示是否需要先保存，begin 和 end 指示了活跃区间。

真正的活跃变量分析在 RegAssignment 类中完成，第一步利用公式'Out' - 'Def' + 'Use'逐语句得到每个语句前后活跃的变量。第二步通过线性扫描算法，得到每个变量的最大活跃区间。并且如果一个变量的活跃区间跨过方法调用，那么对该变量的使用需要保存到栈内。第三步依照线性扫描，完成寄存器分配。

线性扫描算法的时间复杂度为 O(V logR)（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。

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。

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。MIPS 代码可以使用 SPIM 模拟器运行，并且把运行结果和 minijava 对应程序的运行结果进行比较。

6.2 MIPS 语法

MIPS 语法过于复杂冗长，可以到 http://compilers.cs.ucla.edu/cs132/project/mips.html 中查看。但是有一些语法规则是必须了解的：

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

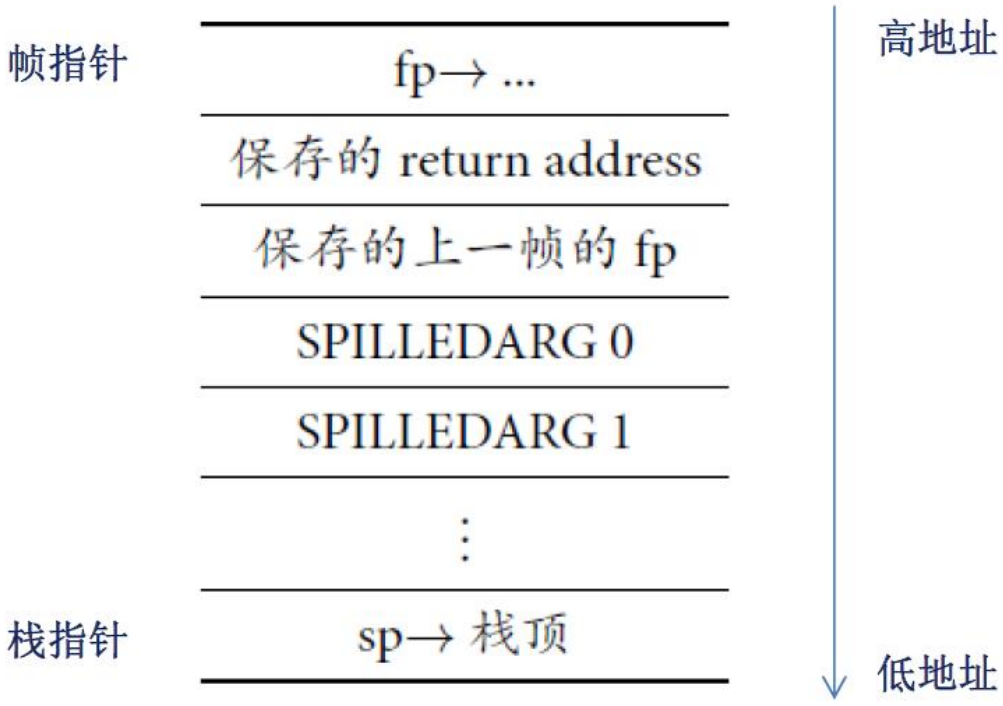
- a0 - a3: 存放向子函数传递的参数
- v0 - v1: 存放子函数调用返回结果，还可用于表达式求值
- s0 - s7: 存放局部变量，在发生函数调用时一般要保存它们的内容
- t0 - t8: 存放临时运算结果，在发生函数调用时不必保存它们的内容
- sp: 栈(stack) 指针
- fp: 帧(frame) 指针
- ra: 返回地址(用于过程调用)

6.2.3 使用到的 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，并将自动将下一指令的地址（返回后的地址）写入$ra，方法调用时需要保存
jalr Rsrc: 使用寄存器的跳转指令，指令的跳转地址在 Rsrc 寄存器中，并将下一指令的地址（返回后的地址）写入$ra，方法调用时需要保存
• 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
• 运行栈结构



子例程开始时：
1、 操作栈分配空间

2、 保存所有用到的 S 寄存器
3、 结束时需要恢复寄存器和栈帧

6.3 代码结构

• kanga2MIPS
– Main.java 主程序
• visitor
- Kanga2MIPS.java 完成翻译的 Visitor

6.4 实现步骤

该部分代码实现非常简单，几乎是逐句翻译即可，用一次 visit 即可完成翻译

6.5 运行栈

栈的维护也中规中矩，唯一值得一提的是开辟的栈空间的计算。由于代码翻译时采用函数调用的参数压栈的方式，因此实际需要开辟的栈空间是 stackNum = stackNum - paramNum + callParamNum + 2;这么大，并且每次传入的参数都放在当前 sp 之上的一段空间里。栈的每次分配是静态的，在函数内部都看不到栈的变化。

7 总结与收获

7.1 代码整合

根目录下的 Main.java 整合上述的五个阶段，生成从 MiniJava 到 MIPS 的的编译器。

7.2 主要收获

• 理解 Visitor 设计模式与抽象语法树
• 了解编译器的具体实现原理和实现步骤
• 学会了线性扫描算法进行寄存器分配
• 锻炼了单例模式和其他设计模式的使用
•明白编译器的分阶段设计的意义（这点可以设想每个阶段具体做了什么，是如何从 minijava 向 MIPS 靠近的，如果没有这个阶段代码翻译上会有什么难点）
• 学会了构造样例 crash 程序，通过阅读代码，并构造样例攻击这个程序验证程序的漏洞并修复。

7.3 课程建议

感觉 SPiglet 到 Kanga 的翻译还是有些紧张，建议把 SPiglet 到 Kanga 和从 Kanga 到 MIPS 合起来，并给 6 周完成，这样对 Kanga 的翻译应该更好。

8 工具测试

1. Typecheck

直接运行，查看是否检测并输出了全部的语法错误。

2. Minijava to Piglet

从 UCLA CS 132 课程网站下载 pgi.jar
使用命令 java -jar pgi.jar < mine.pg 运行生成的 mine.pg，比对输出结果和正确的结果。

3. Piglet to SPiglet

从 UCLA CS 132 课程网站下载 spp.jar 和 pgi.jar
使用命令 java -jar spp.jar < mine.spg，检测是否是规范的 spiglet 语法
使用命令 java -jar pgi.jar < mine.spg 运行生成的 mine.spg，比对输出结果和正确的结果。

4. Spiglet to Kanga

从 UCLA CS 132 课程网站下载 kgi.jar
使用命令 java -jar kgi.jar < mine.kg 运行生成的 mine.kg，比对输出结果和正确的结果。

5. Kanga to MIPS

使用 QtSpim 软件运行.s 后缀的 MIPS 文件，进行调试。

以下是 QtSpim 的运行界面、TreeVisitor.s 的执行结果。