编译实习报告

江振升 1400012781

Github: `https://github.com/JensenJiang/minijava-project`

Monday 19th June, 2017

目录

1 编译器概述

1.1 基本功能

该编译器的目的是将 MiniJava 语言编译成 MIPS 语言，其中经过 MiniJava 的类型检查以及多种中间语言的转换，包括：

1. MiniJava 类型检查
2. MiniJava 到 Piglet
3. Piglet 到 Spiglet
4. Spiglet 到 Kanga
5. Kanga 到 MIPS

该编译器的实现要求是参照 UCLA CS 132 的课程作业要求，有关该编译器的要求细节以及相关的示例程序、语言规范和解释器等请访问 UCLA CS 132 课程主页[1]。

1.2 语言简介

MiniJava MiniJava 是 Java 的一个子集。

- 类型：只支持 Integer, Boolean, Integer Array 以及自定义的类；
- 运算：只支持加法、减法、乘法、小于以及取非；
- 类：允许类的覆盖，但不允许类的重载；
- 系统交互：唯一的可以进行输出的方法是 *System.out.println()*，但参数类型只能为 Integer。

Piglet Piglet 是编译过程中的第一个中间语言，它的要求相对比较简单，方便从 MiniJava 进行转换。

- 变量管理：变量可以存放在无限的临时单元 *TEMP <ID>*，也可以存放在使用 *HALLOCATE <size>* 动态申请的堆空间中；

[1] `http://web.cs.ucla.edu/~palsberg/course/cs132/project.html`

- 变量访问：对于存放在临时单元中的变量，会自动对临时单元进行求值；存放在堆空间中的变量，需要用 *HLOAD* 和 *HSTORE* 进行存取；
- 过程调用：参数会自动存放在 *TEMP 0, TEMP 1, ...* 中，并且过程退出后自动恢复。

SPiglet SPiglet 是 Piglet 的一个子集，与 Piglet 相比，它对允许使用的表达式的形式进行了约束：

1. 不再允许 *StmtExp* 作为表达式使用，即需要进行展开处理；
2. 一些地方不能直接使用 *Exp*，而要使用形式更简单的 *SimpleExp* 和 *TEMP*。

Kanga Kanga 是一门面向 MIPS 的中间语言，从这一步开始不再采用临时单元（或称虚拟寄存器），而是采用真实的寄存器，并同时引入了栈的操作。同时，过程调用中的参数传递以及返回值传递需要自己管理。

MIPS MIPS 是一个由 MIPS Technologies 设计开发的采取精简指令集的处理器架构，关于 MIPS 的信息请参考维基百科上的内容[2]或网络上的相关资料，此处不再赘述。

1.3 编译器特点

- 可以完成从 MiniJava 到 MIPS 的翻译，也可以一步步地进行到中间语言的翻译；
- 同一个类的 DTable 共享同一块内存空间；
- 使用了线性扫描算法进行寄存器分配。

[2] `https://en.wikipedia.org/wiki/MIPS_architecture`

2 编译器设计

2.1 概要设计

2.1.1 Type Checking of MiniJava

SymbolTable *SymbolTable* 类是用于维护符号表的类，是 *DepthFirstVisitor* 的子类，负责在构建语法树的同时建立符号表。它的主要属性是 *ScopeEntry* 类的实例 *global_entry*，存储的是全局范围的符号表条目。它的其它功能还包括一些建立符号表后的解析与检查工作，例如 *extends_resolve_all* 方法用来解析每个类实例真实对应哪个类。

SymbolTableEntry *SymbolTableEntry* 类是用于维护符号表的条目，除了全局范围以外的实例，对应的是每一个类、方法、类属性或者是局部变量。由于不同的符号有不同的特性，因此 *SymbolTableEntry* 派生了以下的子类：*ScopeEntry* 以及其子类 *GlobalEntry*, *ClassEntry*, *MethodEntry*，和 *NodeEntry* 以及其子类 *IntegerEntry*, *BooleanEntry*, *IntArrayEntry*, *ObjectEntry*。其中，*ScopeEntry* 的实例都有一个 *Hashtable* 的属性，用于存储全局范围、类以及方法中的子符号。

SecondPassChecker *SecondPassChecker* 是 *DepthFirstVisitor* 的子类，用于检查第二次扫描才能检查出的类型错误，例如使用了未定义的方法或变量等。

ExpressionChecker *ExpressionChecker* 是 *DepthFirstVisitor* 的子类，用于检查与表达式相关的类型错误，例如表达式中的变量类型是否与表达式的运算相匹配等。

2.1.2 MiniJava->Piglet

在这一步中，要利用之前建好的符号表的信息，进行第一次中间语言的翻译。在开始翻译前，还要对符号表进行一些处理。例如 *build_vd_tables* 方法就是对当前符号表实例分配 VTable 和 DTable 的内存布局，便于在翻译中直接套用该内存布局。

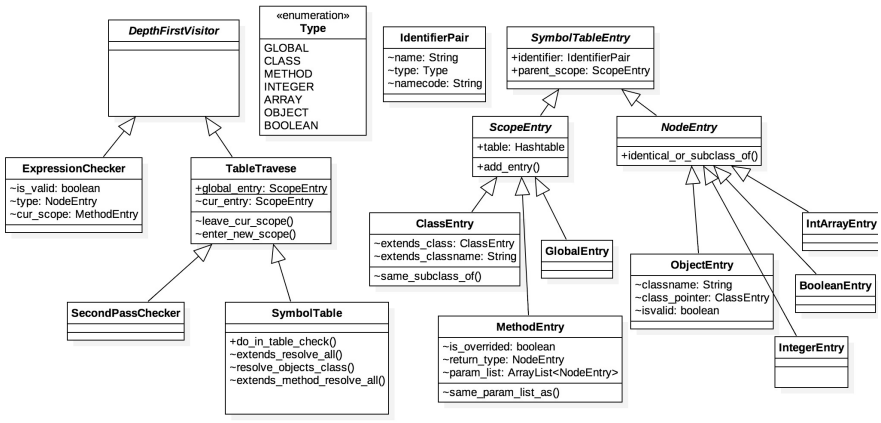

图 1: typecheck 的 UML 图。

PigletBuilder *PigletBuilder* 是 *GJNoArguDepthFirst* 的子类，负责真正翻译的工作。它的属性包括一些翻译的状态，例如 *global_temp_cnt* 存储临时单元的分配情况；*indent* 存储当前翻译代码缩进情况。

PigletFragment *PigletFragment* 是 *PigletBuilder* 的一个内部类，负责直接操纵 Piglet 代码片段的读、写操作。其方法主要是形如 *write_xxx* 的方法，例如 *write_procedure_head* 向当前片段写一个过程的头部。使用 *write_xxx* 方法默认在内部处理好缩进格式，外部不用再处理。

2.1.3 Piglet->Spiglet

SPigletBuilder *SPigletBuilder* 是 *GJNoArguDepthFirst* 的子类，结构与功能类似于 *PigletBuilder*。

SPigletFragment *SPigletFragment* 是 *SPigletBuilder* 的一个内部类，结构与功能类似于 *PigletFragment*。

MaxTempVisitor *MaxTempVisitor* 用于统计 Piglet 代码中使用了多少个临时单元。

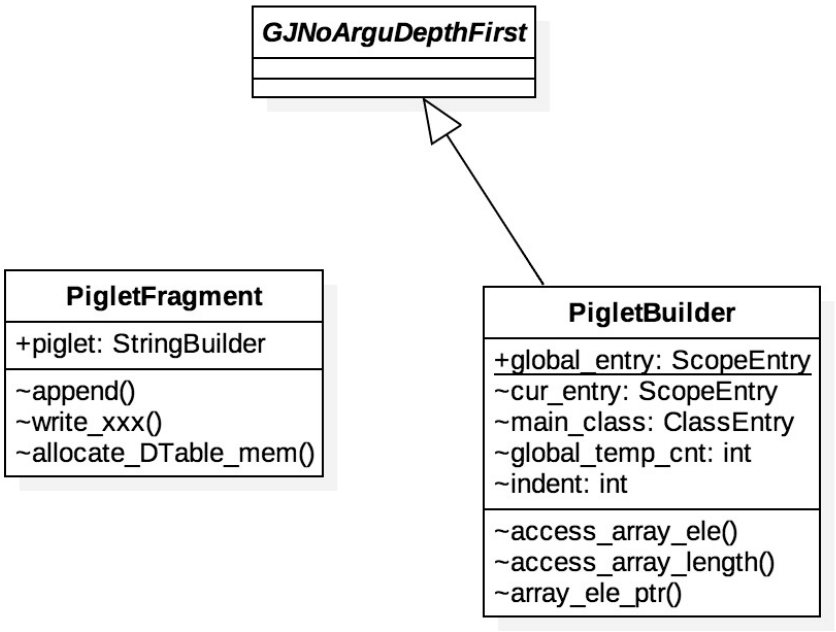

图 2: minijava2piglet 的 UML 图。

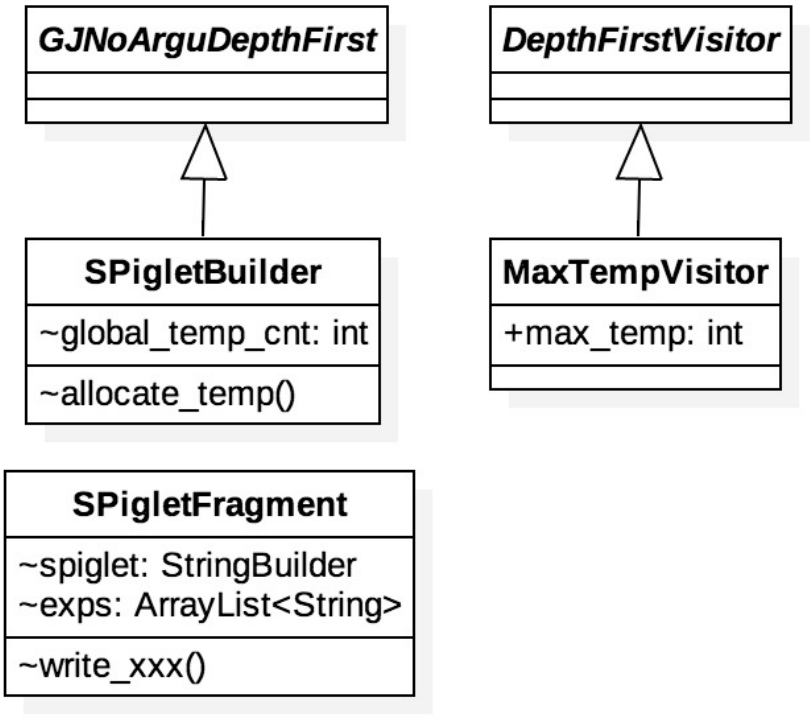

图 3: piglet2spiglet 的 UML 图。

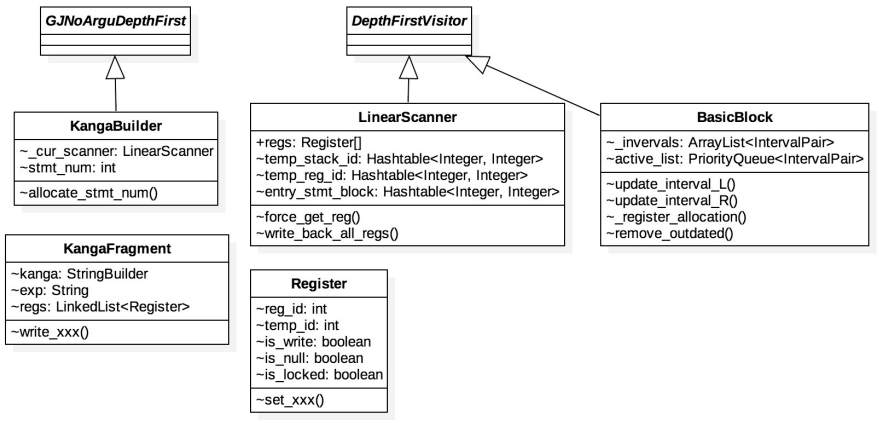

图 4: spiglet2kanga 的 UML 图。

2.1.4 Spiglet->Kanga

KangaBuilder *KangaBuilder* 是 *GJNoArguDepthFirst* 的子类，结构与功能类似于 *PigletBuilder*。

KangaFragment *KangaFragment* 是 *KangaBuilder* 的一个内部类，结构与功能类似于 *PigletFragment*。

LinearScanner *LinearScanner* 是 *DepthFirstVisitor* 的子类，负责对一组语句（通常是一个过程）建立流图，并保存这个过程类临时单元到栈空间、寄存器的映射。在翻译过程中，还负责寄存器相关信息的操作，例如 *force_get_reg* 方法用于为一个临时单元强制获得一个寄存器。

BasicBlock *BasicBlock* 是 *LinearScanner* 的一个内部类，是 *DepthFirstVisitor* 的子类，负责保存流图中基本块的信息以及线性扫描算法所需要的信息，例如 *_intervals* 保存所有临时单元的 *active interval*。同时，在翻译过程中，负责根据线性扫描算法进行通用寄存器的分配，例如 *_register_allocation* 就是为某一个 *active interval* 对应的临时单元分配一个寄存器。

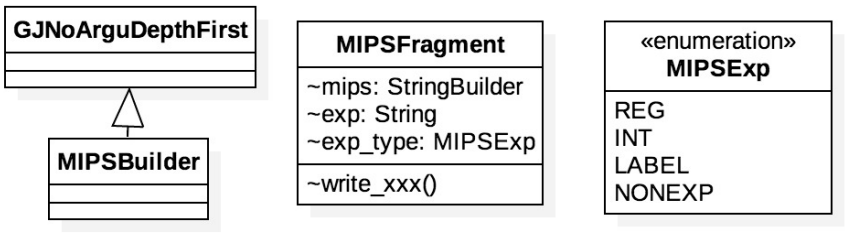

图 5: kanga2mips 的 UML 图。

2.1.5 Kanga->MIPS

MIPSBuilder *MIPSBuilder* 是 *GJNoArguDepthFirst* 的子类，结构与功能类似于 *PigletBuilder*。

MIPSFragment *MIPSFragment* 是 *MIPSBuilder* 的一个内部类，结构与功能类似于 *PigletFragment*。

2.2 详细设计

2.2.1 类型检查

在 MiniJava 的类型检查中，主要需要检查以下几种类型错误：

- 使用未定义的类、变量、方法；
- 重复定义了类、变量、方法；
- 用于判断的表达式必须是 *boolean* 类型；
- 操作数相关：+、-、*、< 的操作数必须是整数；
- 方法参数类型不匹配，参数个数不匹配，类型与方法声明的返回值类型不匹配；
- 类的循环继承；
- 不允许方法重载。

在我的设计中，我将类型检查分为三步：

1. first pass: 即主要为了建立符号表而进行的第一趟扫描；
2. in-table check: 即建立好符号表后，对符号表进行进一步处理时进行的检查；
3. second pass: 即进行第二趟扫描，并根据符号表的信息进行剩下的类型检查。

first pass 在 first pass 中主要进行的操作是建立符号表，详细过程与设计在下一小节中再叙述。在这一步中，要处理的类型错误有：类、变量、方法的重复定义。

in-table check 在 in-table check 中，先要进行的是继承关系的处理。在 first pass 中，我们只记下了继承的类的字符串，因此要先将字符串解析成类符号表项，并同时检查是否有使用未定义类的情况。然后，用有向无环图中找环的方式，检查是否有类的循环继承的错误。

其后，要解析所有类变量其真实对应的类符号表项，这是因为在 first pass 中，我们只记下了类名的字符串。在这一步中，同时检查是否使用未定义类的情况。

最后，对继承关系中继承的类属性与类方法进行处理，同时要处理隐藏与覆盖的问题。此时要处理继承后可能出现的方法重载的问题。当这一步完成后，符号表就真正地建好了。

second pass 在 second pass 中，要进行的是利用已经建好的符号表的信息，对源代码进行第二次扫描，并对剩下的类型错误进行检查。这一步要处理的类型错误有：使用未定义的变量与方法、与表达式相关的错误。其中，与表达式相关的错误，主要是要做表达式的递归检查，即一步步地检查是否有子表达式不符合操作数的要求，或者在赋值、传参中，表达式是否匹配要求的类型。在本编译器中，实现了对子类类型的匹配支持。

2.2.2 符号表

符号表的数据结构设计是一个四级的结构，上一级到一下级是一个包含的关系：

1. 第一级：*GlobalEntry*，可以认为是一个全局范围，一个符号表中只有一个实例；
2. 第二级：*ClassEntry*，对应 MiniJava 中的类；
3. 第三级：*MethodEntry* 与 *NodeEntry*，对应类中的类方法与类属性；
4. 第四级：*NodeEntry*，对应方法中局部变量。

在 *SymbolTable* 中有一个类变量 *cur_entry*，是 *ScopeEntry* 的一个实例，在遍历语法树的过程中，当出现进入或返回到一个全局范围（对应 *GlobalEntry*）、类（对应 *ClassEntry*）、方法（对应 *MethodEntry*）时，都要更新这个变量。

2.3 VTable 与 DTable

在总体设计上，采取 VTable 与 DTable 分离，其中 VTable 的第一个单元存放指向 DTable 的指针。这样设计的目的是：

1. VTable 和 DTable 分离，使得同一个类的不同实例可以共享 DTable 的内存空间；
2. 使 VTable 在形式上与数组类似，方便编码。

但同时，这也使得访问 DTable 时需要多一跳的计算。

而在 VTable 于 DTable 内部的内存布局上，要考虑到 MiniJava 类继承中的隐藏和覆盖的问题。

VTable 与隐藏 隐藏指的是，在子类中定义了父类的同名属性时，在子类中实际同时存在着子类和父类的同名属性，判断访问的是哪一个属性要根据编译时变量的类型来决定。

由于这与类的属性相关，因此要在 VTable 的设计中体现出来。在本编译器的设计中，一个类的 VTable 的内存布局是父类的属性在前，子类的属性在后。在符号表中记录每个属性在 VTable 中的位置，编译时根据变量的类型决定使用哪个符号表项，并以此决定访问的内存地址。

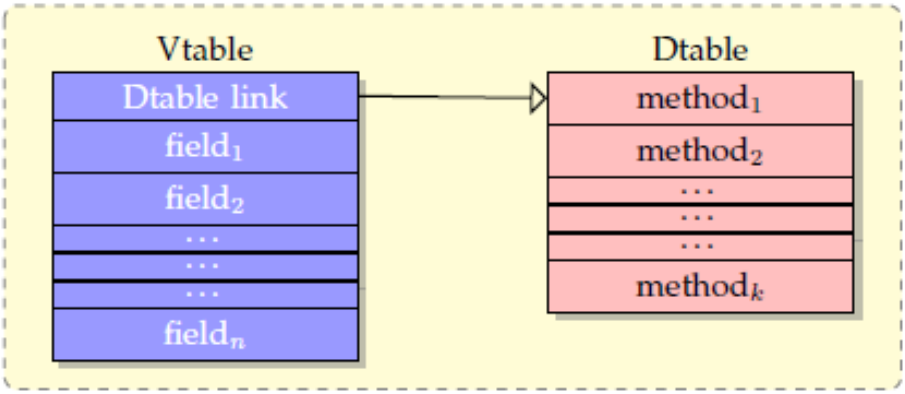

图 6: VTable 与 DTable 的总体设计。

DTable 与覆盖 覆盖指的是，在子类中定义了父类的同签名的方法时，子类的方法要覆盖父类中的同签名方法，即子类中只存在着子类中定义的该方法。判断访问的是哪一个方法时，要根据运行时类型来决定，即如果是一个父类实例，则调用父类中的方法，如果是一个子类实例，则调用子类中的方法。

由于这与类的方法相关，因此要在 DTable 的设计中体现出来。在本编译器的设计中，一个类的 DTable 的内存布局也是父类的方法在前，子类的方法在后。但当出现覆盖的情况时，子类并不是在 DTable 中创建一个新的单元，而是覆盖父类同签名方法的内存单元。这样，子类和父类的 DTable 内存布局中，同签名方法的内存单元只有一个，而且是在同一位置的。那么在运行时，由于子类和父类的实例持有不同的 DTable，就会根据实例的类型调用不同的方法，起到覆盖的作用。

2.4 寄存器分配算法

寄存器的分配方法采用的是 Massimiliano Poletto 和 Vivek Sarkar 提出的 Linear Scan Register Allocation[3]。该方法便于实现，而且执行速度快，与图染色法相比寄存器的利用效率没有下降太多。该算法在网络上能搜索到相关信息，此处不再赘述。

[3] http://dl.acm.org/citation.cfm?doid=330249.330250

3 编译器实现

3.1 工具软件的简介

3.1.1 JavaCC

JavaCC[4]是 Java Compiler Compiler 的缩写，是一个基于 Java 的语法分析器生成器，可以认为是 flex 的 Java 版本。它读入一个符合 JavaCC 规范的语法规则，并产生一个可以识别该语法规则的 Java 程序。本编译器使用的是 JavaCC 5.0 版本。

3.1.2 JTB

JTB[5]是 Java TreeBuilder 的缩写，是一个配合 JavaCC 使用的基于 Java 语言的语法树生成器。它读入一个符合 JavaCC 规范的语法规则，并产生:

- 使用 Visitor 模式的语法树单元;
- Visitor 模式的相关类，例如 *DepthFirstVisitor*、*GJDepthFirst* 等;
- 一个 JavaCC 语法规范，用于给 JavaCC 生成语法分析器。

使用 Visitor 模式可以方便地扩展已有的代码，并且让代码更加清晰易读。

本编译器使用的是 JTB 1.3.2 版本。

3.1.3 Piglet/Spiglet/Kanga Interpreter and Parser

这三门语言都是为了编译教学而编写的中间语言，它们的解释器或者分析器都能在网站上[6]找到。

我觉得需要一提的是关于 *PRINT* 的表现以及整体内存空间的排布。在 Piglet 解释器中，无论是 Label 还是堆空间地址（*HALLOCATE* 的返回值），*PRINT* 的结果都是一个整数，并且可以使用该整数去访问对应的堆空间；但在 Kanga 中，*PRINT* 的结果都是经过修饰隐藏的结果，并且通过整数也无法访问对应的堆空间。可见这两个解释器在内存方面的实现上有一些不连贯的地方。

[4] `https://javacc.org`

[5] `http://compilers.cs.ucla.edu/jtb/`

[6] `http://compilers.cs.ucla.edu/cs132/setup.html`

另外，这些解释器在异常处理上也做得不够完善，有一些 Java 的空指针异常并没有捕捉，使得在出错时会直接打印出解释器的异常信息，而没有给出关于输入代码的信息（例如出错时解析到的行号等），这都给代码调试带来了不便。

3.1.4 Spim

Spim[7]是一个开源的 MIPS32 的模拟器，用来模拟解释 MIPS 代码，并提供一些调试工具，例如寄存器值的显示、单步执行等。

3.2 阶段编码细节

3.2.1 类型检查

在设计环节提到过，我将类型检查分为三个阶段进行，因此在实际编码时，编程语义上也是按照三个阶段划分的。

first pass first pass 的主要工作是建立符号表，而建立符号表的主要工作除了 Visitor 遍历输入代码（这是 JavaCC 和 JTB 已经实现好的）外，就是向对应的 *Hashtable* 中添加符号表项。这个过程在实现时被集中到 *SymbolTableEntry* 的初始化时执行。

```
1 public SymbolTableEntry(IdentifierPair id, ScopeEntry parent){
2     this.identifier = id;
3     this.parent_scope = parent;
4     if(parent != null) parent.add_entry(id, this);
5 }
```

in-table check in-table check 在设计时也是分为三个过程进行的，因此在实现的编程语义上也是按照这三个过程进行的。

```
1 public void do_in_table_check(){
2     extends_resolve_all();
3     resolve_objects_class();
```

[7] http://spimsimulator.sourceforge.net

```
    extends_method_resolve_all();
}
```

second pass 在 second pass 中处理的类型错误主要是与表达式相关的，由于存在括号表达式，因此表达式的处理使用递归的方式会比较自然，递归返回当前处理表达式的状态。因此，我在实现时专门创建了一个 *ExpresionChecker* 来处理。

```
public class ExpressionChecker extends DepthFirstVisitor {
    boolean is_valid;
    NodeEntry type;
    MethodEntry cur_scope;
    ExpressionChecker(MethodEntry cur){
        this.cur_scope = cur;
        this.is_valid = false;
    }
    void wrong_exp(){
        this.is_valid = false;
        this.type = null;
    }
    /* Visitors */
}
```

另外在参数传递时，表达式有多个，为了处理这种情况，我创建了 *ExpressionListChecker* 来处理。它的属性与 *ExpresionChecker* 类似，但是用 *ArrayList* 存放的。

3.2.2 代码翻译

在代码翻译环节，主要要做的事情可以归为以下两类：遍历源代码，对目标代码进行写操作。为了让代码逻辑更加清晰，我创建了两个类来分别处理这些事情，并把直接写操作集中起来，便于维护。由于对各个翻译过程而言都是类似的，这里选择 piglet2spiglet 作为例子。

Builder *Builder* 都是 *GJNoArguDepthFirst<Fragment>* 的子类，主要是用来遍历语法树，并把子节点返回的 *Fragment* 按照语法规则合并起来，再返回到父节点进一步处理。

在 *Builder* 中还需要保存一些编译过程中的全局状态，例如在 *SPiglet-Builder* 中，*global_temp_cnt* 用来保存当前临时单元的分配情况，*indent* 用来保存当前的缩进级别。在其它 *Builder* 中有类似的状态变量。另外，也有一些操纵这些状态的方法，例如 *allocate_temp*。

还有一些与 *Fragment* 实例无关的，但也是用来生成代码变量的方法也被放在了 *Builder* 中。在 *SPigletBuilder* 中，这些都是不带换行的行内表达式，例如 *HALLOCATE*。

Fragment *Fragment* 是 *Builder* 的内部类，是用来封装和处理目标语言代码片段的类。以 *SpigletFragment* 为例，它的主要属性有：用来构建当前代码片段的 *spiglet*，以及用来处理是表达式类型的 *exps*。

为了语义上的清晰以及便于调试，在实现上将所有对 *spiglet* 的写操作都统一到 *Fragment* 中提供的方法进行。首先对于代码片段的连接操作，提供了 *append* 方法进行操作。另外，对于写特定语句到代码片段中的操作，提供了形如 *write_xxx* 的方法。以 *SpigletFragment* 为例，*write_cjump* 可以向当前代码片段中写入一个 *CJUMP* 语句。使用 *write_xxx* 方法不用考虑缩进和换行的问题，在方法内部已经解决。

3.2.3 线性扫描算法

在 spiglet2kanga 过程中，需要考虑寄存器分配的问题，但抛开的这个问题，翻译部分其实也是可以用 *Builder* 和 *Fragment* 的框架来解决的。因此，只要另外再新建一个框架来负责响应寄存器分配的请求即可。在我的实现中，我创建了 *LinearScanner* 和 *BasicBlock* 这两个类来解决这个问题。

LinearScanner *LinearScanner* 是 *DepthFirstVisitor*，是用来在翻译源代码前先遍历一次语法树建立一个过程的流图。由于流图包含了若干个基本块，因此把基本块的构建放到 *BasicBlock* 去做，*LinearScanner* 中只处理一些流图中全局的状态，例如 *temp_stack_id* 存放的是临时单元到栈的映射，*temp_reg_id* 存放的是临时单元到当前所用寄存器的映射。

另外，虽然寄存器分配的请求应该由某一个基本块来处理，但一个时刻

只对应一个基本块，因此可以在 *LinearScanner* 中存放当前基本块的变量，并提供一些基本块的方法封装以便于外部使用。

BasicBlock *BasicBlock* 是 *LinearScanner* 的内部类，是用来执行线性扫描算法的主体类。它的属性包含了线性扫描算法要维护的数据结构，例如 *_intervals* 用来存储所有的 *active intervals*，*active_list* 则维护了一个优先队列，用来快速淘汰结束时间过长的 *active intervals*。

执行寄存器分配的最主要方法是 *_register_allocation*，另外还有一些辅助算法执行的方法，例如 *remove_outdated* 用来淘汰时间过长的 *active intervals*。

```
/* Block Register Allocation */
void _register_allocation(int cur_stmt_num, KangaBuilder.KangaFragment kf) {
    boolean has_clean = false;
    while(!this._intervals.isEmpty() && this._intervals.get(0).get_first() <= cur_stmt_
        /* Remove the outdated registers */
        if(!has_clean) {
            this.remove_outdated(cur_stmt_num, kf);
            has_clean = true;
        }
        IntervalPair cur_interval = this._intervals.remove(0);
        /* check if TEMP has taken up a register */
        if(temp_reg_id.get(cur_interval.temp_id) != null) continue;
        /* exists available register */
        if(!_avail_general_regs.isEmpty()) {
            int reg_id = _avail_general_regs.poll();
            _allocate_nomadic_reg(cur_interval.temp_id, reg_id);
            used_regs.add(reg_id);
            active_list.add(cur_interval);
        }
                else {
                    IntervalPair first = this.active_list.peek();
                    if(first.get_second() > cur_interval.get_second()) {
                        /* recycle */
```

```
                        Integer reg_id = temp_reg_id.get(first.temp_id);
                        write_back_reg(regs[reg_id], kf);
                        active_list.remove(first);

                        /* add new */
                        _allocate_reg(cur_interval.temp_id, reg_id);
                        active_list.add(cur_interval);
                    }
                }
            }
        }
        else {
            IntervalPair first = this.active_list.peek();
            if(first.get_second() > cur_interval.get_second()) {
                /* recycle */
                Integer reg_id = temp_reg_id.get(first.temp_id);
                write_back_reg(regs[reg_id], kf);
                active_list.remove(first);
                /* add new */
                _allocate_reg(cur_interval.temp_id, reg_id);
                active_list.add(cur_interval);
            }
        }
    }
}
```

另外，在翻译进行的时候，应该根据当前执行到的语句切换 *LinearScanner* 中的当前基本块。

3.2.4 MIPS 栈空间布局

在最后一步 kanga2mips 中，引入了一块连续的栈空间，这就需要编译器去处理过程调用时栈帧的初始化，以及从过程返回时如何恢复原过程的栈帧。要实现上述的功能，就需要在当前过程的栈帧中存放相关的数据。

图 7: MIPS 栈空间布局。

如图 7，当从一个过程返回时，先把 *$sp* 设为 *$fp*-4，然后把当前的 *$ra* 先存到一个临时寄存器里面，然后恢复上一个过程的 *$fp* 与 *$ra*，再根据刚才存好的返回地址跳转回上一个过程下一步应该执行的指令的地址。

3.3 测试

3.3.1 测试用例的构造

构造测试用例时我是针对某一个单元的功能进行构造的。例如在做 spiglet2kanga 的翻译过程中，曾经遇到过解释器说执行超过一百万条指令，超出允许范围的错误。因为官方所给的测试用例都不可能执行这么多步，因此可以判断是进入了死循环。但是出错的测试用例代码太长，不便于在 kanga 代码中调试，因此我用 minijava 写了一个简单的只有 while 循环的程序，并将其一步步翻译为 kanga 代码。

3.3.2 测试效果

在自己构造的用例中，可以看到，存放记录循环计数的寄存器出现了两个，其中一个被更新了，但在写回栈的时候，两个寄存器都被写回去了，而且没被更新的写回晚于被更新的写回，因此导致循环计数器一直为初始值。从这个情况可以判断出，是寄存器分配的步骤出了问题，这样就可以定位到编译器的出错位置了。

另外，在定位 kanga 代码的出错部分时，还可以运用 *PRINT* 指令来进行输出调试。kanga 解释器对于 *PRINT* 的输出有更加丰富的信息，例如对于堆空间来说，可以打印出里面的每一个元素是什么，甚至是如果里面的元素也是一个堆空间地址的话，也会把它的内容嵌套地打印出来。我使用输出调试找到了在写回所有寄存器时的一个错误。

4 实习总结

4.1 收获与体会

4.1.1 主要的收获

我在本课程中的收获主要有以下几点：

- 了解并亲身体验到了写一个编译器的主要步骤，包括从设计到代码实现；
- 对 Java 语言有进一步的了解；
- 学习到了 Visitor 模式的使用。

4.1.2 学习过程的难点

由于编译技术这门课前半个学期都是在讲前端的内容，但本课程的实习是不涉及前端的内容的，因此会觉得编译技术的关于前端的内容无法得到实践，而且编译实习的内容也因为进度远快于编译技术而一开始觉得不知道从何下手。

另外，说到本课程五个作业的难度，我个人认为是 spiglet2kanga > typecheck ⩾ minijava2piglet > kanga2mips > piglet2spiglet。其中，typecheck 的符号表的设计尤为关键，它很大程度上影响了 minijava2piglet 的难写程度。另外，spiglet2kanga 中的寄存器分配是一个比较难调试的部分。

还有一点是关于因为不了解后面语言的特性而导致前面的设计无法正常地在后面的语言中实现。我在实现 minijava2piglet 的过程中，对于同一个类的 DTable 使用了同一片内存空间。但这样的话就得有一个全局能访问的地方存下这些 DTable 的指针。在 piglet 于 spiglet 时，临时单元都是全局可见的，因此没有什么问题；但到了 kanga 后，栈是局部可见的，这就使得 DTable 的指针无法自然地在全局可访问了。我最后的解决方法是把 DTable 的指针作为过程调用的第一个参数传入，这样就可以付出一定的内存代价使其全局可见了。

4.2 对课程的建议

4.2.1 实习内容的建议

建议可以适度引入一些代码优化相关的内容，例如在 spiglet2kanga 部分的时候，可以介绍一些如何优化寄存器使用次数的方法，并测试在运行某个样例时访问栈的次数，以此来鼓励同学们去优化寄存器分配的方式。

4.2.2 对老师讲解内容与方式的建议

由于本课程进度要快于编译技术，我觉得老师可以在每个作业之前给同学们简单讲讲这方面的理论基础以及现在的主流做法，并提供一些进一步了解的阅读材料，例如编译技术课本的相关章节，或者是一些相关论文等。

另外，我认为实习报告其实可以提前给定大概框架，并告知同学可以在编写程序的时候同步写实习报告。因为我发现留到期末再写报告的话，前面的一些细节记得不太清楚了，需要自己回看一下代码才能想起来。而且，一时间要写这么多内容的话，可能在写的时候也会不留意间选择性地忽略了一些内容，使得报告的完整性降低。