编译实习报告

1500012727 赵子栋

目录

一 概述

1. 编译器概述

编译器是将“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为：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]

本课程参考了 UCLA CS 132，目标是实现一个将 **MiniJava** 翻译为 **MIPS** 的编译器。

在此报告中，涉及到的语言的详细BNF定义和描述均可以在UCLA网站上获取：MiniJava，Piglet，sPiglet，Kanga，MIPS。

2. 基本功能

本编译器实现的功能有：

1）类型检查：检查 MiniJava 代码中可能出现的类型错误、变量声明错误、方法声明错误、继承错误等。

2）MiniJava → sPiglet：将 MiniJava 代码翻译为 sPiglet 中间代码。实现面向对象到面向过程的转换，实现高级语言变量、操作符、分支结构到三地址代码（中间代码）的转换。

（注：在本次作业中，我直接将 MiniJava 翻译为了 sPiglet，跳过了 Piglet 的生成。因此，没有实现 Piglet → sPiglet 的转换。）

3）sPiglet → Kanga：将 sPiglet 中间代码翻译为 Kanga 中间代码。实现寄存器的分配，实现部分栈结构的构建，实现死代码消除、冗余分支消除等代码优化功能。

4）Kanga → MIPS：将 Kanga 中间代码翻译为 MIPS 汇编语言。实现三地址代码到体系结构指令集的映射，实现栈结构的维护。

5）以上的每个步骤都实现了对输入代码的词法和语法检查。

最终，本编译器将 MiniJava 翻译为 MIPS，并保证翻译后的汇编代码的执行结果与通过 javac 编译的 MiniJava 程序的执行结果完全相同（注：MiniJava 没有程序输入接口，因此正确性检查只需考虑程序的输出）。

[1] 百度百科，编译器

3. 特点

本编译器的特点有：

1） 是一个将 MiniJava 翻译为 MIPS 的编译器。

2） 使用 Java 编写（便于调用词法和语法分析器接口），大量使用 Visitor。

3） 使用 JTB 和 JavaCC 作为词法和语法分析器的前端实现。

4） 中间语言为 sPiglet 和 Kanga。

本编译器的**额外**特点有：

1） 能报告类型检查错误发生的行、列。

2） 能报告所有至少存在的类型检查错误，而不是一个。

3） 类型检查额外检查了错误。

4） 可扩展性好，容易实现变量扩展（加入 `void`、`String`、其他数组类型等），容易实现变量即时声明（即在代码块中声明变量，而无须预先声明全部变量），容易实现分支类型扩展（加入 `for`，`return` 等）。

5） 寄存器分配采用基本块划分-活跃变量分析算法、图染色算法。

6） 实现了死代码消除、冗余标签消除。

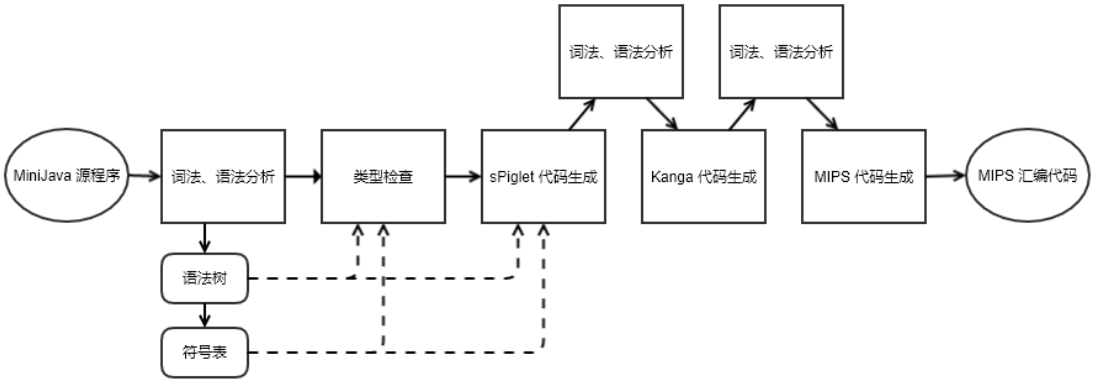
4. 实现代码

本次作业的完整代码、部分测试样例以及正确性检查的批处理文件均托管在 GitHub 上。代码的详细使用方法请参阅 README.md 文件。

二 编译器设计

1. 总体布局

在本部分中，主要谈论的是编译器的设计细节，包含了类框架设计和算法设计，不包含具体的代码。从总体来看，本编译器的具体流程由下图所示：



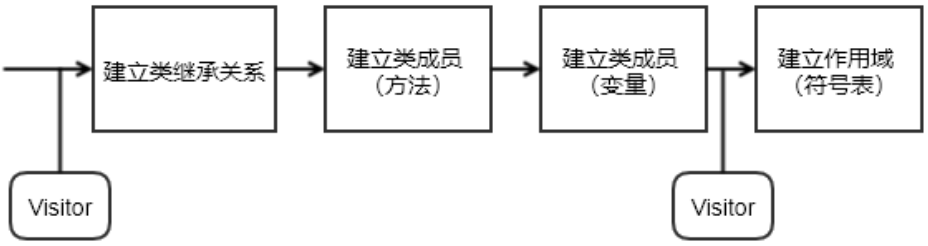
每一步的词法和语法分析都会生成新的语法树。特别地，在类型检查和 sPiglet 代码生成阶段需要用到符号表。

2. 类型检查

2.1 流程

类型检查可分为两个阶段：建立符号表和错误检查。由于错误检查贯穿于整个流程，且与 MiniJava 语义高度相关，因此在本部分只大体描述符号表的建立过程，错误检查的具体实现和例子留至第三部分。

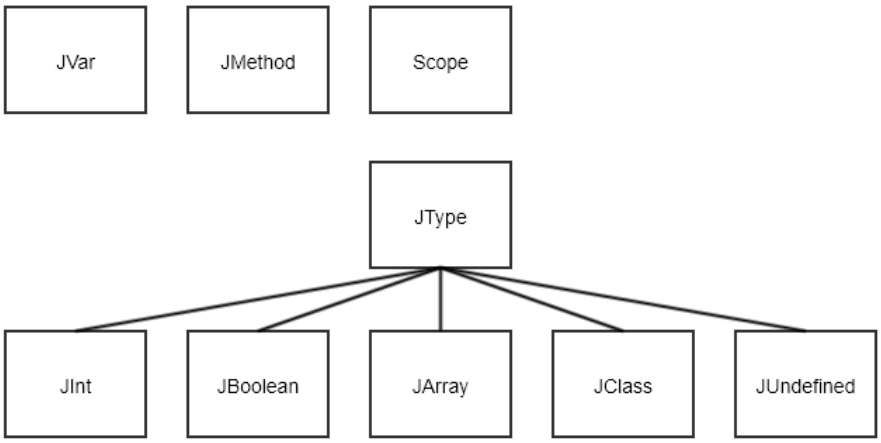
符号表的建立过程主要通过 Visitor 扫描语法树实现。由于类和方法均可在被使用之后声明，因此一次扫描语法树是不能正确建立符号表的。因此，要分步处理类声明和方法声明。类型检查的流程图如下（已完成词法和语法分析）：



2.2 框架描述

2.2.1 符号表描述

符号表主要由以下几个类构成。其相互间的**继承**关系如下（有连线才代表继承）：



其中，每个类的定义如下：

1） JType

此为 MiniJava **变量类型**的抽象描述。这与之后的对 MiniJava **变量**的抽象描述：`JVar` 类相区别。之所以将变量类型、变量抽象为两个类，而不是仅仅将变量本身抽象为一个类，是因为注意到了变量类型本身一经声明（无论是 built-in 还是用户声明的，如 `class`），就不会被改变；对于源代码的符号分析是基于变量的，同一种类型可以声明多个变量，从运行时的观点来看，变量是在动态的变化，且还会有临时变量（匿名）的出现。将二者分开，有助于类型检查和临时寄存器分配的编写。

注意：对于每个 MiniJava 变量类型而言，其都是 `JType` 的一个**实例**。比如，对于 `int` 类型，在编译器中，会被视作一个 `JInt` 对象（`JInt` 为 `JType` 的子类），是通过 `new JInt()` 创建并保存的。（因此对于 built-in 类型而言，其有且只有一个实例对象，可视为常量）

`JType` 及其子类应该具有的功能至少有：返回本类型名；返回本类型实例应占用的字节大小；判断其他类型与本类型之间的赋值合法性。

2） JVar

此为 MiniJava **变量**的抽象描述。如前所述，变量拥有变量类型，它们相互之间是成员关系而不是继承关系。应该把变量看作变量类型的实例。除了用户定义的变量，在表达式中生成的临时变量也应该视为 `JVar` 的实例。

其应该具有的功能至少有：返回本变量标识符（若是临时变量，可为空）；返回本变量的类型；返回在源代码中的声明位置（便于错误检查的行列定位）；维护本变量的位置（寄存器/内存），便于 sPiglet 代码生成。

3） JMethod

此为方法的抽象描述。在最开始，我曾考虑将 `JMethod` 也视为变量类型（受 c 函数指针的影响）。但后来发现在 Java 中，方法严格地按照 `()` 区分，使得其可以与变量、类名重名，且不存在函数指针。因此，最终将 `JMethod` 独立为一个类。

其应该具有的功能至少有：返回方法名；返回所属类；维护返回值类型和参数类型。

4） Scope

此为作用域的抽象描述。这是符号表的核心，也是符号表的具体实现。因为不满 MiniJava 必须预先声明所有变量的限制，我抽象了 `Scope` 类。这样做的好处是，将符号的所有声明和查找都交由 `Scope` 处理，在遍历语法树时，只需要传递 `Scope` 实例，在出现变量声明时（此时的变量声明当然是任意位置的），就能动态加入到 `Scope` 中去。于是，日后只需要修改 MiniJava 语法树，就能轻易实现变量即时声明，大大提高了可扩展性。

其应该具有的功能至少有：维护作用域之间的包含关系；能够向上层作用域转移局部变量的查询请求。

5） JUndefined

此为缺失类型的变量类型描述，是 `JType` 的子类。受 JavaScript 影响，我抽象了这个类。其用处在于，能够容忍类型错误，继续进行分析。例如，即使使用了未声明的变量 a，类型检查也会为其创建一个变量类型为 `JUndefined` 的 `JVar` 实例，这样就与没有错误时的行为保持一致，简化了报告错误的代码编写，也增强了代码简洁性。

其应该具有的功能至少有：实现 `JType` 的所有功能。

6） JClass

此为用户自定义的类的抽象描述，是 `JType` 的子类。其本质上没有特别的地方，但作为面向对象语言的核心和 sPiglet 代码生成逻辑的中心，其代码占比较大。

其应该具有的功能至少有：实现 `JType` 的所有功能；维护继承关系；维护并查找成员变量；维护并查找成员方法；明确与父类之间的方法覆盖关系。

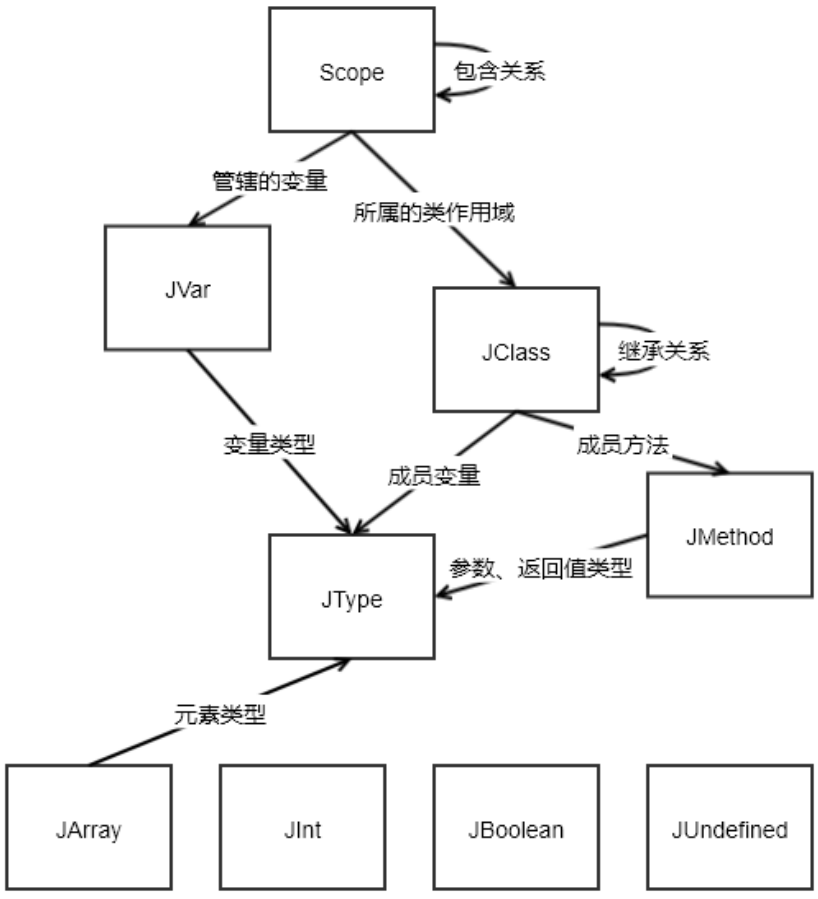
7）JInt，JBoolean，JArray

这些均为 MiniJava 的 built-in 类型，是 `JType` 的子类。除 `JArray` 需要维护元素类型以外，只需实现 `JType` 的所有功能即可。

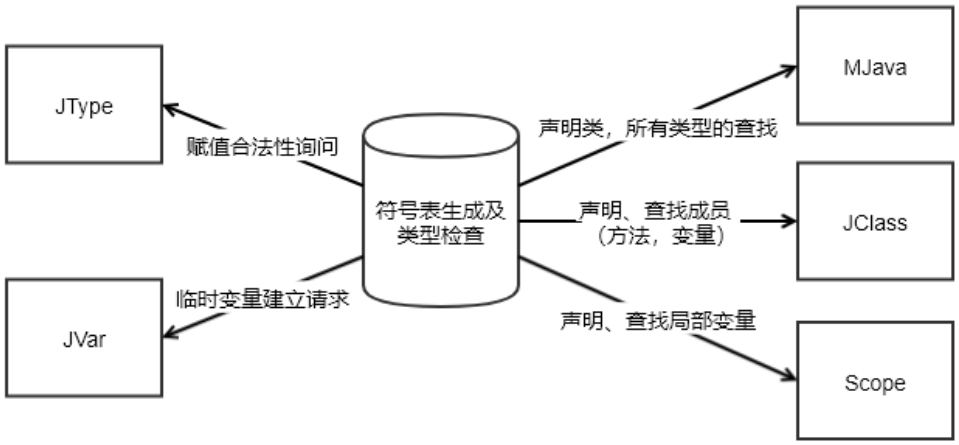
除了这些类以外，还有一个 `MJava` 类作为符号表的总入口，维护所有 `JType` 实例。

2.2.2 符号表结构

根据前一小节的描述，不难构建出符号表的结构。其中，类之间的**成员**关系（某个类的成员的类型为另一个类）如下图所示：



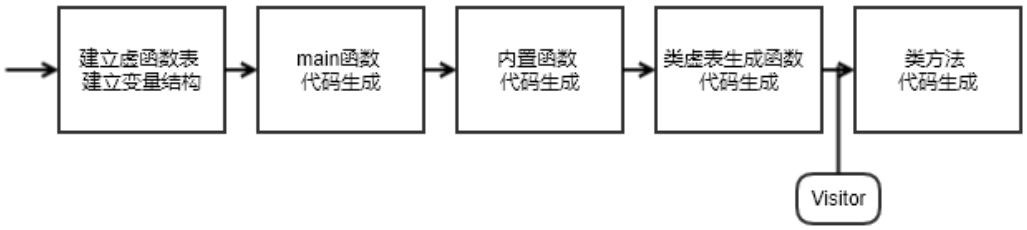
它们需要实现的**接口**如下图所示:



3. sPiglet 代码生成

3.1 流程

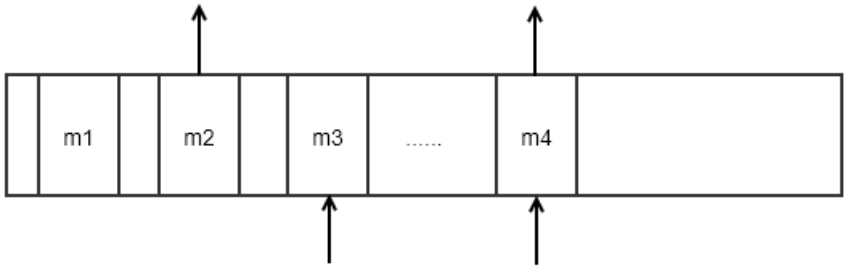
由于 sPiglet 是从面向对象到面向过程语言的翻译，对类概念的消除是必须的。在此过程中，较难的是实现子类方法对父类方法的覆盖，即多态。受 C++中对多态的实现的影响，我采用的是为每个类的实例绑定一张虚函数表的方法。出于消除 sPiglet 代码冗余的考虑，又将虚表的生成过程封装为一个生成函数（下图第 4 步）。于是，本阶段的流程如下图所示:



3.2 框架描述

3.2.1 虚函数表建立

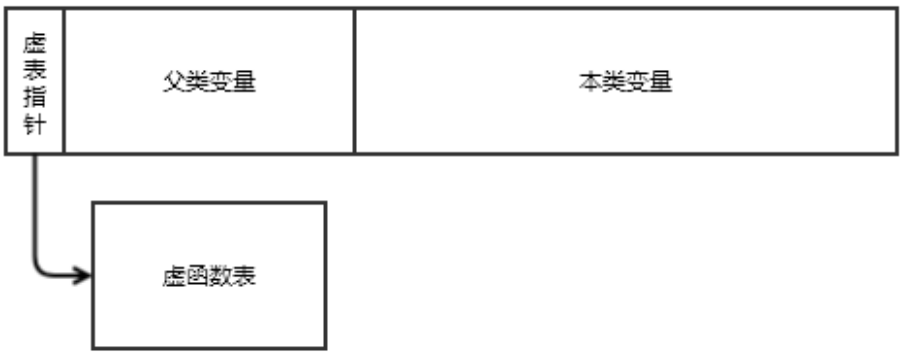
虚函数表的建立算法较为简单直接。在建立了类的继承关系以后，扫描每个类的每个方法，如果存在覆盖，则向上标记所有父类的被覆盖的此方法。扫描完成后，对于每个类，其方法状态有如下四种(自由,覆盖了父类某方法,被子类某方法覆盖,既被覆盖也覆盖其他):



显然只有 m1 是自由的，其他方法均是虚函数。为了实现多态，则所有在同一条覆盖链上的方法都应该拥有相同的偏移。此时，编译器只需要通过确定的偏移调用虚表中的函数，就能实现多态。显然，对于偏移量的分配优先级应为 m2=m4>m3>m1（实际上，m2 和 m4 由父类函数偏移决定），且总体分配顺序应为先父类再子类。

3.2.2 变量结构建立

由于变量的访问是根据编译时类型的，因此不难构造变量存储结构如下图所示：

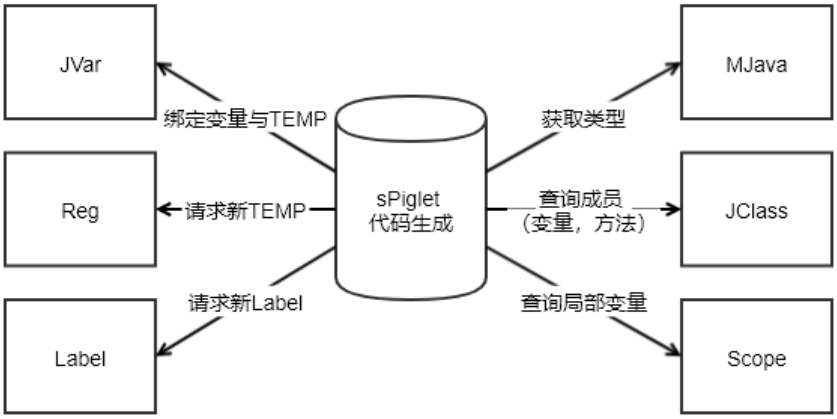


3.2.3 代码生成

由于仍在 MiniJava 语言架构中，因此 sPiglet 代码生成仍需要使用类型检查的语法树和符号表。在此阶段，只新增了两个类：`Code` 和 `CodeGenerator`。`Code` 作用与具体实现技巧有关，在第三部分会说明。`CodeGenerator` 是一个 Visitor，负责实现每个语法生成式的 sPiglet 代码逻辑。`CodeGenerator` 实现的是代码逻辑转换。

因为 sPiglet 代码没有函数嵌套，因此采用将代码片段作为返回值、由调用者生成代码的方法是不合适的。我采用的是编译技术课上提及的 on-the-fly 代码生成方法。在这种方法中，被调用者直接将代码片段输出，而无须告知调用者。因此，它要求调用者小心地实现调用逻辑，保证调用顺序的正确性。

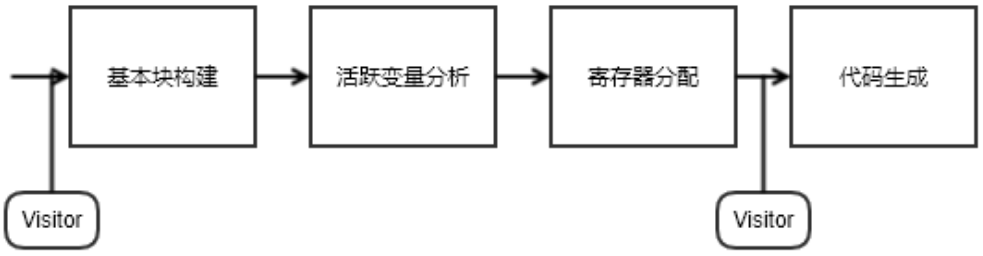
在此基础上，还加入了 `Reg` 和 `Label` 两个类，以实现全局的 `TEMP`（即 sPiglet 中的寄存器）和标签分配。于是，在 sPiglet 代码生成的过程中，类需要实现的调用如下图所示：



4. Kanga 代码生成

4.1 流程

对于 Kanga 代码生成，我采用的是基本块划分-活跃变量分析算法和图染色算法。其大致的流程如下图所示：



4.2 框架描述

4.2.1 基本块构建

基本块构建又可分为划分基本块和构建控制流图（Control Flow Graph）两个部分。在具体实现中，我将基本块和控制流图分别抽象为 `Block` 和 `Graph` 两个类。它们的具体算法如下所述：

基本块划分：

1） 确定基本块入口

a） 若这是第一条指令

b） 若这条指令拥有标签

c） 若这条指令紧跟在 `JUMP` 或 `CJUMP` 或 `ERROR` 的后面

2）每个基本块入口唯一对应一个基本块

构建控制流图：

1） 每个基本块的前驱和后继按照控制语句的语义建立

2） 添加入口基本块，唯一后继为第一个基本块，没有前驱

3） 添加出口基本块，前驱最后一个基本块，和以 `ERROR` 为结尾的基本块，没有后继

4.2.2 活跃变量分析

活跃变量分析（Live Variable Analysis），是编译技术课上提及的一种寄存器分配和死代码消除算法。其核心是维护每个基本块的 IN，OUT，USE 和 DEF 集合。它们的定义如下：

1） IN：在基本块的入口处活跃的变量集合

2） OUT：在基本块的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

显然，对于一个基本块来说，USE 和 DEF 都是常量，可以预先计算。而 IN 和 OUT 集合则采用不动点（Fix Point）算法来计算，伪代码如下：

```
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 };
```

4.2.3 寄存器分配

寄存器分配是 Kanga 代码生成的重点。我将其分为以下三个部分：

1） 确定寄存器语义

根据 ICS 课上学到的知识，寄存器应分为 Caller Save 和 Callee Save 两种。结合 Kanga 寄存器的实际，确定寄存器语义如下：

s0-s7	通用；在函数调用的上下文中保证值不变；由被调用者负责保存
t0-t9	通用；在函数调用的上下文中可能会被修改
a0-a3	用于传递前 4 个函数参数
v0	函数返回值；临时存放溢出变量
v1	临时存放溢出变量

2） 构建寄存器冲突图（Register Interference Graph）

在完成了活跃变量分析之后，构建寄存器冲突图较为简单：对于每一个基本块，从下往上逐条扫描语句，将出现在同一条语句的活跃集合中的所有 `TEMP`，两两之间构造冲突边即可。需要注意的是，由于存在函数调用的情况，需要将跨越函数调用（即 `CALL` 语句）的活跃变量做特殊标记，以保证它们只会被分配到 `s0-s7` 寄存器或直接溢出。

3） 图染色（Graph coloring）

图染色的算法较为简单，其流程如下（设可用寄存器数目为 k）：

a） 找到一个度数小于 k-1 的节点，将其删除并入栈

b） 若无法找到，则选择一个度数最大的节点，将其删除，并溢出

c） 直到图为空，则将节点依次出栈，保证在原图中与相邻节点染色不同

注意：如前所述，染色时要预先处理特殊标记的变量，此时可用寄存器只有 8 个；而对于一般变量的染色，可用寄存器有 18 个。

4.2.4 代码优化

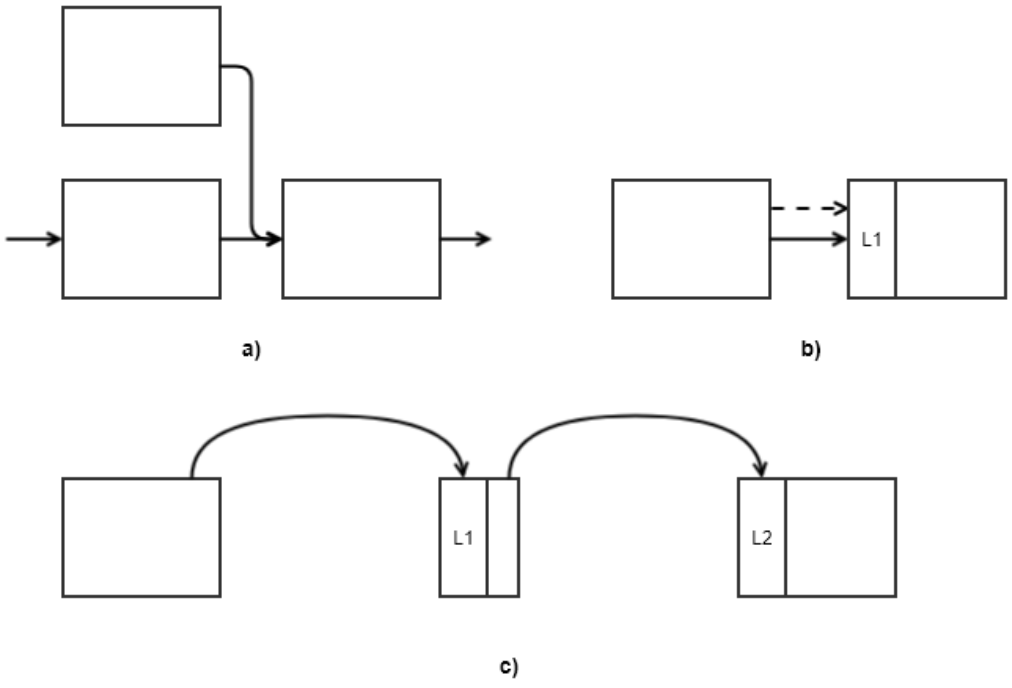
在分配好寄存器以后，代码生成是十分简单的。出于对 Kanga 代码优雅性的考虑，**额外**实现了 Kanga 代码优化。其主要内容有如下两部分：

1） 死代码消除

死代码即无用的代码。具体到 sPiglet 代码中，表现为定义的 `TEMP` 变量在后续没有被使用。注意到只有 `MOVE` 和 `HLOAD` 语句才会定义变量，因此只需要在扫描到这两个语句时，判断其定义的 `TEMP` 是否出现在活跃变量集合中。若不出现，则将整个语句删除。显然，某条语句的删除可能会引发更多的死代码消除，因此需要使用类似不动点的算法来尽可能多地进行语句的删除。

2） 冗余标签/基本块消除

伴随着死代码的消除，一些标签也会变得冗余。由于时间的限制，我没有对所有可能的冗余标签情况做讨论，只考虑了如下几种最为常见的情况：



a） 没有前驱的基本块（不包括入口基本块）。这样的基本块是不可达的，应当消除。

b） 对于某个标签，其没有被作为跳转指令的目标，或者使用它的跳转指令就是上一条指令。这样的标签显然是冗余的，应当消除。

c） 对于某个基本块(如图中央的基本块)，其**只**含有跳转指令，且只会跳转至某个确定的标签（对于`CJUMP`指令也可能成立，比如其跳转的目标就是下一条指令）。那么应当将此基本块的标签与其目标标签合并。在某些情况下，还可以直接删除此基本块。

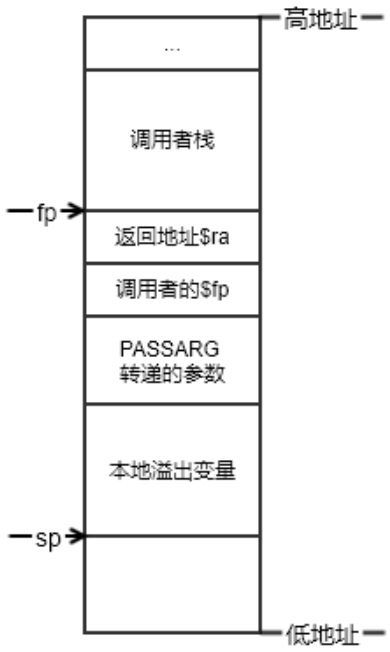
5. MIPS 代码生成

5.1 流程

MIPS 代码生成是本编译器中最为简单的部分，使用 Visitor 扫描语法树一遍即可完成代码的生成。由于流程过于简单，因此便不做流程图展示。

5.2 框架描述

MIPS 代码生成的主要内容是栈结构的构建和栈指针的维护。根据 ICS 课上所学的知识，构建栈结构如下图所示（从`$sp`到`$fp`为一个有效的运行时栈）：



由于在 Kanga 代码中已经统一了`PASSARG`和溢出变量，因此可以直接通过`$sp`加上偏移的方式访问。在函数开始和结束时维护好返回地址和`$fp`、`$sp`的值，代码生成就大体完成了。

三、编译器实现

1. 工具软件

1）Visual Studio Code

VSCode 是一款功能强大的代码编辑器，界面美观，有速览定义、转到定义等便捷功能，调试功能也十分强大。因此，我把它作为本次作业编写的 IDE。需要注意的是，其配置较为复杂。本次作业对于 VSCode 的配置随代码附在 GitHub 上。

2）JavaCC 和 JTB

本编译器采用的词法和语法分析器是 JavaCC 和 JTB。只需输入符合 BNF 的`.jj`文件，即可自动地生成分析器代码。在实际调用中，通过`Goal()`方法就能访问语法树的根节点。以 MiniJava 分析器为例，使用如下命令，即可自动生成分析器代码：

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

3）Piglet Interpreter

此为 Piglet 代码（也是 sPiglet 代码）的解释器。将 Piglet 代码作为标准输入提供，即可得到运行结果。其缺点有：

a）函数直接传递的参数不能超过 20 个。这其实是相当没有必要的限制，因为后续的 Kanga 已经规定了函数直接传递的参数不能多于 4 个。

b）当执行的语句超过约 3000 条时，解释器会因自身栈溢出而报错。即使在 MiniJava 代码中没有超过 3000 次的循环，但由于数组申请时需要初始化为 0，因此在申请元素超过 600 的数组时，也会引发上述错误。

4）sPiglet Parser

此为检查 Piglet 代码是否为 sPiglet 的分析器。由于本次作业直接生成了 sPiglet 代码，因此基本不需要使用到这个工具。

5）Kanga Interpreter

此为 Kanga 代码的解释器。将 Kanga 代码作为标准输入提供，即可得到运行结果。其缺点有如下两点：

a）擅自划分指针类型和数值类型。在某个表达式中，一旦出现了指针（特指通过 `HALLOCATE` 获得的值），整个表达式都会被认为是指针类型。而在 `CJUMP` 中，如果其使用的寄存器是指针类型，就会报错。这导致了指针的比较成为不可能。

b）不允许指针减法。所有对于指针类型的减都相当于无效（但不报错）。这导致了内存组织的方式十分受限。例如，一开始我将数组长度保存在-1 位置（即-1 位置指向的是 `HALLOCATE` 返回的值，但对外展现的数组起始地址为 0 位置），导致数组长度无法访问。

6）QtSpim

此为 Windows 系统下的 MIPS 模拟器。通过 UI 界面载入 MIPS 汇编代码，运行后得到输出。缺点较为明显，其既不能支持命令行运行，UI 界面也十分简陋且不友好。

2. 各阶段编码细节

2.1 类型检查

1）Visitor 实现

根据前一部分设计的框架，执行类型检查的 Visitor 的参数类型为 `Scope`，返回值为 `JVar`。之所以将返回值类型设为 `JVar`，是受到了 C++中表达式均有类型的影响。这样的设计也便于嵌套的类型检查和之后的代码生成。

于是，定义了 Vistor 如下：

```
public class ScopeBuilder extends GJDepthFirst<JVar, Scope> {
    …
```

2）变量声明与查询

符号表的建立过程较为庞杂，在此以变量的声明与查询为例。

当 Visitor 访问到 `VarDeclaration` 生成式时，向 `Scope` 发送声明请求：

```
public JVar visit(VarDeclaration n, Scope scope) {
    scope.declare(MJava.queryType(n.f0.f0.choice), n.f1);
    …
```

在 `Scope` 中，首先检查是否有重定义错误（由于允许方法中局部变量对类成员变量的覆盖，此时的重定义检查局限于方法体中）：

```
public void declare(JType t, Identifier id) {
    String sid = id.f0.toString();

    JVar v = queryVarOnlyFunc(id);
    if (v != null) {
        ErrorHandler.send("Dupilcate variable " + sid, id);
        return;
    }
    …
```

如果没有错误，就将符号和 `JVar` 间的映射加入到维护的 `Map` 中：

```
vars.put(sid, new JVar(id, t));
```

当 Visitor 需要通过符号获取一个变量时，向 `Scope` 发送查询请求：

```
public JVar queryVar(Identifier id) {
    // look up id in Func body
    JVar t = queryVarOnlyFunc(id);
    if (t != null)
        return t;

    // look up id in Class & Father Class
    return owner.queryVar(id);
}
```

`Scope` 首先在方法体内部查找，如果无法找到，就向高一级的类作用域转移查询请求。如果符号也不对应任何一个类成员变量，就返回一个类型为 `JUndefined` 的 `JVar` 临时变量。于是，Visitor 负责检查“使用未声明变量”的错误：

```
public JVar visit(Identifier n, Scope scope) {
    JVar a = scope.queryVar(n);
    if (a.Type() instanceof JUndefined) {
        ErrorHandler.send("Undefined variable " + a.Id(), n);
    }
    …
```

3）类型不匹配的检查

错误检查贯穿于整个流程。限于篇幅，在此以一个常见的错误——类型不匹配作为错误检查的例子。

根据框架设计，`JType`的所有子类都应该实现判断其他类型与本类型之间的赋值合法性的功能。这种抽象的目的是基于类型转换和可扩展性的考虑。假如日后允许`boolean`类型自动转换为`int`类型，或加入新的`void`类型，只需更改判断的逻辑即可轻松实现。在代码中，判断赋值合法性的逻辑由`Assignable()`方法实现。如，在`JInt`中：

```
public boolean Assignable(JType a, boolean report, Node n) {
    if (a instanceof JInt)
        return true;
    else if (report)
        ErrorHandler.typeMismatch(this, a, n);
    return false;
}
```

只允许`JInt`类型作为合法的赋值对象；而在`JClass`中：

```
public boolean Assignable(JType c, boolean report, Node n) {
    if (!(c instanceof JClass)) {
        if (report)
            ErrorHandler.typeMismatch(this, c, n);
        return false;
    }
    JClass p = (JClass) c;
    while (p != null) {
        if (p == this) {
            return true;
        }

        p = p.father;
    }
    …
```

除了赋值的对象必须为`JClass`以外，还必须保证其是本类或本类的子类。由于错误会被自动报告，因此在 Vistor 中，优雅地调用`Assignable()`方法，就能检查类型不匹配错误。

例如，检查`if`语句的表达式是否为`boolean`类型，只需对下层语法树节点的返回值`exp`（这是一个`JVar`对象）作如下调用即可：

```
public JVar visit(IfStatement n, Scope scope) {
    JVar exp = n.f2.accept(this, scope);
    MJava.Boolean().Assignable(exp.Type(), true, n);
    …
```

4）死代码/死循环的检查

作为类型检查的 Bonus，我针对一些常见情况，做了死代码/死循环的检查。其逻辑十分简单，即判断 `if` 或 `while` 分支是否使用了恒为真或假的值作为表达式。例如在 `if` 分支中：

```
        if (exp.Val() == JBoolean.True()) {
            ErrorHandler.warn("Dead code", n.f6);
        } else if (exp.Val() == JBoolean.False()) {
            ErrorHandler.warn("Dead code", n.f4);
        }
```

可能会产生两种死代码的情况。同理对于 `while` 则会可能会产生死代码或死循坏的情况。

判断表达式是否恒为真或假的逻辑是，为所有真假字面值对应的 `JVar` 绑定一个真假值：

```
    public JVar visit(TrueLiteral n, Scope scope) {
        return new JVar(n, MJava.Boolean()).assign(JBoolean.True());
    }
```

当做`&&`运算时，若存在一个已知的恒为真或假的值，就将结果绑定为真或假（注意此时也可能会产生死代码，因为另一个参与运算的表达式相当于无效）：

```
    public JVar visit(AndExpression n, Scope scope) {
        …
        int val = JBoolean.Unknown();
        if (a.Val() == JBoolean.False()) {
            ErrorHandler.warn("Dead code", n.f2);
            val = JBoolean.False();
        } else if (b.Val() == JBoolean.False()) {
            ErrorHandler.warn("Dead code", n.f0);
            val = JBoolean.False();
        }
        …
```

5）整型字面值过大的检查

作为类型检查的 Bonus，我检查了整型字面值过大的情况。由于基于正则表达式的词法分析器无法限制整型字面值的位数，为了避免 sPiglet 运行时的错误，做如下检查：

```
    public JVar visit(IntegerLiteral n, Scope scope) {
        try {
            Integer.parseInt(n.f0.toString());
            return new JVar(n, MJava.Int()).assign();
        } catch (NumberFormatException e) {
            ErrorHandler.send("literal … is out of range", n);
            return new JVar(n, MJava.Int()).assign(); }
    }
```

2.2 sPiglet 代码生成

1）代码输出接口的抽象

由于在之后的编译器阶段，都需要输出代码，因此在实现中，我抽象了一个 `CodeWriter` 类。这样做的好处是能统一代码输出的接口，提升简洁性：

```
public class CodeWriter {
    private static PrintWriter out;
    public static void init(String filename) { … }
    public static void finish() { … }
    public static void emit(String code) { … }
    public static void emit(String code, String prefix, String suffix){…}
}
```

于是，对于特定的代码生成（如 sPiglet），只需继承它并实现语言接口即可：

```
public class Code extends CodeWriter {
    public static void mov(int dreg, String exp) {
        emit("MOVE " + T(dreg) + " " + exp);
    }
    …
```

在代码生成过程中，像这样优雅地调用，就能输出类似 `MOVE TEMP 0 TEMP 1` 的代码：

```
    Code.mov(left.Reg(), right.Exp());
```

2）语义翻译逻辑

语义翻译是高级语言向低级语言翻译的核心。在此以翻译 `if` 语句为例：

```
        exp = n.f2.accept(this, scope);
        Code.jump(f, exp.Reg());
        n.f4.accept(this, scope);
        Code.jump(exit, -1);
        Code.label(f);
        n.f6.accept(this, scope);
        Code.label(exit);
```

为了正确翻译，首先要计算 `if` 语句表达式的值。如前一部分所述，代码生成采用的是 on-the-fly 方法。因此在调用了 `exp = …`之后，计算表达式的值的相应 sPiglet 代码**已经**生成并输出了。这个表达式的值被保存在一个 `TEMP` 寄存器中，寄存器的编号则保存在 `exp` 这个 `JVar` 实例中。

接下来，生成一个 `CJUMP` 语句,条件跳转至 `f(false)`标签；然后生成表达式为真的代码（通过遍历语法树子节点，on-the-fly），小心地生成跳转至 `exit` 的指令;小心地创建 `f` 标签，生成表达式为假的代码；生成 `exit` 标签，作为 `if` 控制块的出口。

由此可见，通过充分利用 on-the-fly 特性、正确处理语义逻辑，就能优雅并正确地生成 sPiglet 代码了。

3）数组的构建与安全性检查

数组的内存结构为：`length` 存储在偏移 0，第 i 个元素存储在 i+1。由于 Java 语义要求对数组进行越界检查，因此需要生成额外的 sPiglet 代码。我将其封装为 `checkOutOfIndex()`。由于越界检查逻辑较为简单，在此不再赘述。

4）一些技巧

为了使生成的 sPiglet 代码更简洁，我将初始化内存块为 0 的过程封装为 `calloc [2]`；将虚函数表的生成过程封装为函数，如类 `A` 的虚表生成过程会被封装为 `new_A [0]`。

需要对每个 MiniJava 方法进行编号，不能简单把 `类名_方法名` 作为 sPiglet 函数的名字。这是因为若在既定义了类 `A_B` 和方法 `C`，又同时定义了类 `A` 和方法 `B_C` 时，就会产生冲突。

采用 on-the-fly 方法，若不经优化，会将常见的 `a = 3` 这样的 MiniJava 语句翻译为：

```
MOVE TEMP 21 3
MOVE TEMP 22 TEMP 21
```

其中，`TEMP 21` 作为临时变量，在后面的语句都没有出现。显然这样会造成大量的冗余，对随后的寄存器分析和代码优化带来麻烦。解决的方法之一是在随后的代码优化中实现常量传播，缺点是编写较为复杂。我采用的方法是通过 `setLiter()`调用，标记每个整型字面值所对应的临时 `JVar` 实例：

```
return new JVar(n, MJava.Int()).bind(treg).setLiter(n.f0.toString());
```

虽然这个 `JVar` 实例绑定了寄存器 `treg`，但还没有生成关于 `treg` 的任何代码；若这个临时变量可以直接作为字面值出现在 sPiglet 代码中（也即 BNF 中作为生成式 `SimpleExp` 时），那么就调用 `JVar.Exp()`方法；否则调用普通的 `JVar.Reg()`方法，此时才会生成加载的代码。

例如，翻译 `t = 4 + 5`，`a` 为从语法树节点“4”返回的 `JVar` 实例，绑定了 `TEMP 23`；`b` 为从语法树节点“5”返回的 `JVar` 实例，绑定了 `TEMP 24`；结果需要保存在 `TEMP 25` 中。

经过如下调用：

```
Code.plus(rreg, a.Reg(), b.Exp());
```

就会生成如下 sPiglet 代码（消除了 `MOVE TEMP 24 5`）：

```
MOVE TEMP 23 4
MOVE TEMP 25 PLUS TEMP 23 5
```

2.3 Kanga 代码生成

1）主要算法的实现

在编译器设计部分，已经描述了 Kanga 代码生成中使用到的各类算法，它们的具体代码实现较为平凡。在此列举各算法在代码中的具体实现的方法名，代码细节则不再赘述。

基本块构建	`Graph.stmt()`
基本块活跃变量集合构建	`Block.scan()`
活跃变量分析（不动点算法）	`Block.FixedPoint()`
寄存器冲突边构建	`Graph.addEdge()`
寄存器图染色	`Graph.preColorRIG()`, `Graph.colorRIG()`
冗余基本块/标签消除	`Block.checkDeadBlock()`, `Block.checkLabel()`

2）一些技巧

由于 sPiglet 的语法规则较为奇特，其生成的 sPiglet 抽象语法树对实际编程十分不友好。即使完成了寄存器分配，生成 Kanga 代码也没有想象中的简单。例如，在 sPiglet 中，将 `MOVE` 语句强行分析为 `MoveStmt ::= "MOVE" Temp Exp`，导致即便翻译类似 `MOVE TEMP 0 TEMP 1` 的语句，都没有办法在 Visitor 的一次调用中完成。为了使代码看起来更为简洁和优雅，针对此情况，我引入了一个 buffer，将 `MOVE` 语句的生成划分为两步：

```
public static void mov(int dreg) {
    emit("MOVE " + REG[dreg] + " " + buf);
    buf = "";
}
public static void mov_(String exp) {
    buf = exp;
}
```

在 `MoveStmt` 节点的处理中，只知道 `MOVE` 的目的寄存器 `dreg`。其先调用：

```
n.f2.accept(this, g); // mov_ will be invoked here
```

交由知道 `MOVE` 源寄存器的 `SimpleExp` 节点将源寄存器写入到 buffer 中：

```
Code.mov_(g.getExp(n));
```

然后在 `MoveStmt` 节点中调用 `mov()`，此时才将 Kanga 代码输出：

```
Code.mov(dreg);
```

这样就能优雅地完成 `MOVE` 语句的翻译了。

2.4 MIPS 代码生成

1）栈结构的维护

栈结构已在编译器设计部分详细描述。具体实现为在函数起始维护栈结构:

```
        Code.sw(sp, -4, ra);
        Code.sw(sp, -8, fp);
        Code.mov(fp, sp);
        Code.addi(sp, sp, -slots * 4);
```

在函数结束维护栈结构:

```
        Code.addi(sp, sp, slots * 4);
        Code.lw(ra, fp, -4);
        Code.lw(fp, fp, -8);
        Code.jr(ra);
```

通过`$fp`和对应偏移的方式访问函数参数和本地溢出变量:

```
        lw(R(n.f1), fp, (-3 - I(n.f2.f1)) * 4);
```

通过`$sp`和对应偏移的方式向被调用函数传递参数:

```
        Code.sw(sp, (-2 - I(n.f1)) * 4, R(n.f2));
```

2）系统调用的封装

对于 Kanga 代码中的`PRINT`和`ERROR`指令，分别封装成了 MIPS 函数：`println` 和`err`。其逻辑较为简单，具体的 MIPS 代码可以参见编译器生成的 MIPS 汇编文件，在此不再赘述。

3）立即数的问题

MIPS 只支持 16 位立即数，超出限制的需要使用`li`指令预先加载。因此，在进行指令翻译特别是`BinOp`翻译时，需要判断非 16 位立即数的情况。预先加载所使用的临时寄存器也需要慎重考虑。一般来讲，可以使用`$v0`或`$v1`做为临时寄存器。然而，因为本编译器提供分阶段的接口，因此不排除用户提供的 Kanga 代码中出现如`MOVE v0 PLUS v1 80000`这样的情况（此时 80000 不能加载到`$v0`或`$v1`中）。且不能排除选取的临时寄存器其实是活跃的情况（即在之后会被使用）。因此，最为保守的方法应该是，选取与本指令使用到的寄存器都不同的一个临时寄存器（如在上例中，可选取`$t9`），并在上下文中将这个寄存器保存在栈中，保证这个寄存器不会因为这个语句使得值发生变化。

这种做法也适用于普通的内存访问(如访问数组的某个较大的下标)。然而，对于`PASSARG`而言，如果其参数超过了 16 位（虽然很难发生），则不能使用这种方法。因为`PASSARG`使用了`$sp`，会与将临时寄存器保存在栈中的逻辑相冲突。目前还没有想出一种优雅的解决方法。

3. 测试

3.1 样例构造与测试方法

测试样例基本来源于 UCLA 网站上的八个标准程序，包括了 MiniJava 部分、sPiglet 部分和 Kanga 代码。以下是额外样例的构造：

1）在测试类型检查错误时，根据 Java 的一些特性自己构造了一些样例（由于 MiniJava 语法格式限制很大，编写十分麻烦，因此没有手动构造过多样例）。

2）在检查 sPiglet → Kanga 时，构造了一些较为复杂的基本块流图模式，以测试冗余标签/基本块消除的性能。

3) 在测试 sPiglet → Kanga 以及 Kanga → MIPS 时，参考了一些 GitHub 上的 MiniJava 编译器的输出（即将 UCLA 的标准 MiniJava 程序做输入，把其编译得到的 sPiglet 代码作为测试 sPiglet → Kanga 的样例）。

测试时，使用到了 UCLA 提供的解释器。方法为编写批处理文件，自动运行所有样例的测试并检查输出的一致性。具体的批处理文件在源代码的`/judge`目录下。

3.2 测试中发现的错误

1）数组长度的错误

在 UCLA 的八个标准程序中，都没有针对数组长度`.length`的使用，导致实际上编译器代码中关于数组长度的部分没有被测试。一开始，我错误地将申请内存的长度参数视为了`.length`（真正的数组长度应该是申请内存长度的值减 4），且在很长的一段时间里都没有发现。最终是在封装数组安全性检查的过程时才发现了这个错误。本错误的检出也说明测试样例的全面性是非常重要的。

2）标签错误

在生成 sPiglet 代码时，我采用的标签是`L*`的形式。于是在随后的 Kanga 代码生成时，我想当然地认为输入的标签都是`L*`的形式，于是做了特殊的处理。然而，由于提供了分阶段的接口，用户提供的 Kanga 代码的标签可能是任意形式的（只需符合词法规则），因此只能做通用的处理。这一点直到我开始采用上述第三点中的样例时才发现。

本错误的检出提高了我的警觉性。比如在随后的 MIPS 代码生成中关于立即数的处理中，我便采用了最保守的做法，保证程序的正确性（如 2.4 中所述）。

四 实习总结

1. 编译器总结

基于编译技术课的基础知识以及对课程要求的理解，通过五个阶段的分步实现，最终完成了一个将 MiniJava 翻译为 MIPS 的编译器。除了实现课程要求的基础功能，还额外实现了额外错误检查、死代码消除、冗余标签消除等功能。实现代码可扩展性好，风格较为简洁，健壮性好。使用 GitHub 托管编译器实现的所有代码，包和类的组织结构较为直观。

2. 收获与体会

学习了编译技术课以后，我对于编译器的实际架构并没有太过深入的认识，对语法制导翻译的理解也只停留在书面作业上。在独立完整地写出一个编译器之后，我对编译技术课上的各项知识有了不一样的体会。编译实习课与理论课的相互呼应，诠释了“纸上得来终觉浅，绝知此事要躬行”的道理。

其次，在学习本课程之前，我还没有编写过任何 Java 程序。通过编写编译器这样一个庞大的工程，我的语言学习能力得到了加强，对于 Java 的基本特性特别是 Visitor 有了深入的了解，同时也提升了调试代码和检查错误的能力。

最后，通过编写编译器，我还获得了充足的成就感。在刚接触编程时，我认为编译器十分神秘，惊叹于它能“读懂”代码的能力。现在我真正地实现了一个能够使用的编译器，虽然功能十分有限，但它足以带给我充足的成就感，同时也增强了我面对未知与困难的勇气。

3. 对课程的建议

本次实习主要涉及的是理论课上关于后端生成的部分，词法分析和语法分析的部分全部交由 JavaCC 和 JTB 自动生成。我认为，编译器的前端设计也是十分重要的，其核心思想有十分广泛的应用。因此，如果本课程能够涉及一些词法或语法分析的部分会更好（从本课程受限于对应的 UCLA 课程架构的事实来看，做到这一点不太容易）。

其次，提交代码的评测反馈较为缓慢，如果能较为及时地得到前一阶段代码实现的反馈，就能更好地发现错误，完善下一阶段代码的编写。

1500012727 赵子栋