MiniJava Report

杨晨阳 #1700012770
袁野 #1700012821

2020 年 6 月 12 日

目录

1 Type Check

1.1 Minijava 介绍

Minijava 是 java 语言的一个子集，只保留了 java 的一些核心要素以便于学生写课程作业。具体来说，有以下明显的限制。详细内容，可以参考 UCLA 课程网站 minijava 的 BNF 文法。

1. 不允许重载，但允许子类方法对父类方法的覆盖。
2. 基本类型只有 boolean 和 int。
3. Print 语句只能打印 int 变量。
4. 只有 int 数组，因此 e.length 只能作用于 int[] 型变量。
5. 不允许多重继承。
6. 不允许同时声明多个变量。
7. 所有方法都是 public 的；所有域都没有声明访问限制，但文法不允许从外部访问域。

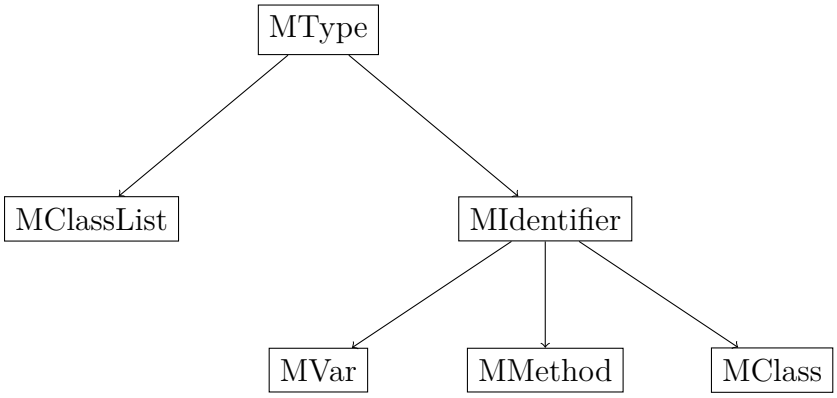
1.2 Visitor 模式

我们利用 Visitor 模式，分离对语法树的定义和对语法树的操作。Visitor 包含对每个需要操作的类的 visit 函数，对语法树而言，也就是每个类型的节点都有一个 visit 函数。我们通过递归调用（例如，对于文法 A->B，有过程 accept(A)->visit(A)->accept(B)->visit(B)），来实现遍历整个语法树。

Visitor 模式的一个好处是，我们可以为每遍 parsing 定义一个 visitor 的 class，使得代码结构分离、干净、统一。具体实现上，我们利用 JavaCC 和 JTB 生成 minijava.jj 对应的语法树，里面便已经包含了 visitor 模式的基本框架。

1.3 构建符号表

符号表需要记录各个类、方法中的成员变量（方法），为后续的类型检查、翻译工作提供遍历。我们的符号表设计参照了课件，继承关系具体如下图。



具体而言，每个类包含的内容如下。

- MType 记录 type（主要为 MVar 使用）。
- MIdentifier 记录 identifier 的 name。
- MMethod 记录所在的类、返回类型、参数列表、局部变量表，以及提供一个实参栈用于类型检查。我们提供 isEqual 函数检查重复出现的方法名是重载还是覆盖，并提供 checkRealArgs 函数检查实参形参类型是否一致。
- MClass 记录方法表、变量表、父类。如果没有明确声明父类，默认为 Object。除了常规接口外，特别地，我们还提供 isChildOf 函数来判断当前类是否是某个类的子类。
- MClassList 记录 String->MClass 的 HashMap，并提供相应读写接口。

我们使用 BuildSymbolTableVisitor (extend GJDepthFirst<MType, MType>) 构建符号表。我们在访问每个类时新建 MClass 对象，并插入 MClassList 中；在访问每个方法时，新建 MMethod 对象，并插入相应的 MClass 中；在访问每个变量声明时，新建 MVar 对象，并插入相应的 MClass/MMethod 中。为了获知当前的 Class/Method，我们使用 argu 参数传递相关信息。特别地，在插入时，我们需要检查是否重复定义，但我们允许父/子类之间方法、变量的覆盖，因此只需在当前类/方法中检查重复定义问题即可。另外，由于 minijava 只允许在类/方法开始申明变量，因此除了类/方法变量之间有覆盖，方法内部不存在作用域问题。

此外，在建立符号表过程中，我们还可以检查类的循环继承问题（A->B->C->A）。事实上，每次出现 ClassExtendsDeclaration 语句时，我们都会记录对应类的父类，亦即加入继承图中的一条边。如果继承图中存在环，当我们插入最后这一条边后，总是可以沿着父类找到这个环，便可以发现循环继承。

1.4 类型检查

我们使用 TypeCheckVisitor 遍历语法树，检查需要构建符号表后才能发现的错误。这些错误主要分为三类。

- 类型不匹配（Type mismatch），需要考察以下情况。

1. if、while 的判断表达式必须是 boolean 型
2. Print 参数必须为整数
3. 数组下标必须是 int 型
4. 赋值表达式左右操作数类型匹配
5. 操作符对应的操作数类型匹配，如 +、*、< 等操作数须为整数。
6. 方法的实际调用（invocation）需要和方法本身的 signature 类型匹配，要注意参数的个数/类型、返回类型的匹配。

- 使用未定义的类/方法/变量。
- 特别地，由于 minijava 不允许重载，我们还需要检查方法的重载问题。我们只需在声明方法时递归检查父类是否有同名方法，而且两个方法的 signature 不同。

在 Visitor 中，我们提供两个统一的函数处理错误。注意到，在处理用户定义的类时，如果 destType 是 inType 的子类，我们也不报错。这对应着赋值时上溯造型的可能性。

```
1   public void handleTypeMismatch(String inType, String destType, String info, int line) {
2       if (inType.equals(destType))
3           return;
4       MClass mClass = MClassList.get(destType);
5       if (mClass != null && mClass.isChildOf(inType))
6           return;
7       String errMsg = "type mismatch (" + info + "): "
8           + inType + ", " + destType + " in line " + line;
9       ErrorPrinter.addError(errMsg);
10  }
11
12  public void handleUndefined(String info, int line) {
13      String errMsg = "undefined " + info + " in line " + line;
14      ErrorPrinter.addError(errMsg);
15  }
```

为了检查类型不匹配问题，我们要求 Expression 语句的返回值是一个 MType 变量，表明该 exp 的类型。注意 Identifier 会返回 name，因此需要在 PrimaryExpression 处进行特殊处理。具体而言，我们在以下场景检查类型不匹配问题。

- 在方法声明时，检查返回的变量类型是否和方法声明的返回类型相同。（本质上是赋值语句左右侧的类型匹配）
- 赋值语句检查左右类型匹配。
- if/while/print 语句中 exp 类型需要符合 minijava 规范。

- &&/</+/-/*/! 对应操作数类型匹配。
- 访问数组、访问数组长度、新建数组实例、数组赋值相应 exp 的类型匹配。
- 调用方法时，实参和形参类型匹配。我们通过将实参加入当前方法的实参列表栈中进行记录，并使用 checkRealArgs 函数检查。注意我们需要使用一个列表栈，而非简单的列表，因为参数中可能存在方法调用，需要区分不同方法的实参。

具体而言，我们在以下场景检查未定义的类/方法/变量。

- 遇到 Type 是 Identifier 的变量，检查 Identifier 是否在 MClassList 中，若不在即出现了未定义的类。
- 对于 MessageSend 语句 a.f(b)，我们需要检查 a 是否为 user-defined class，f 这个方法是否已经定义，注意需要递归检查父类。
- new 类的实例时，检查对应类是否已经定义。
- 特别地，在出现 class A extends B 时，我们需要检查 B 这个类是否已经定义。
- 每次使用变量时，利用符号表检查当前的类/方法中是否声明了这个变量，注意需要递归检查父类。

2 Piglet Translate

2.1 Piglet 介绍

Piglet 是我们遇到的第一个中间语言，也是我们面临的第一个挑战。本次的难点主要在于如何将面向对象语言转化成面向过程的中间代码。此时，所有的成员函数均变为一般的全局函数，所有的变量都变成临时寄存器。同时，还要实现 Java 语言的变量与函数的继承关系、变量的隐藏、方法的覆盖与多态三个特性。另外函数参数大于 20 个时，需要一些额外操作来保存参数。

2.2 Piglet 翻译过程简述

首先，由于 Piglet 代码仍然存在比较复杂的缩进，因此单独使用一个类 PigletPrinter 来打印代码。该类包含一个静态变量 tabNum，代表当然代码块的缩进数，以及一些静态成员函数，用来方便我们打印（例如函数的进入，离开等等）。

在 Piglet 的翻译中，我们使用了 Lab1 中的符号表来获得类之间的继承关系，对于临时变量的分配问题使用了一个全局变量来记录当前过程的下一个临时变量标号，每进入一个过程后更新为 20；labelNum 记录了当前的 label 标号。对于顺序逻辑过程的翻译，只需要把局部变量映射到一个寄存器内。分支循环，数组变量等等在下一节中详细叙述。

2.3 Piglet 翻译难点实现

2.3.1 数组的实现

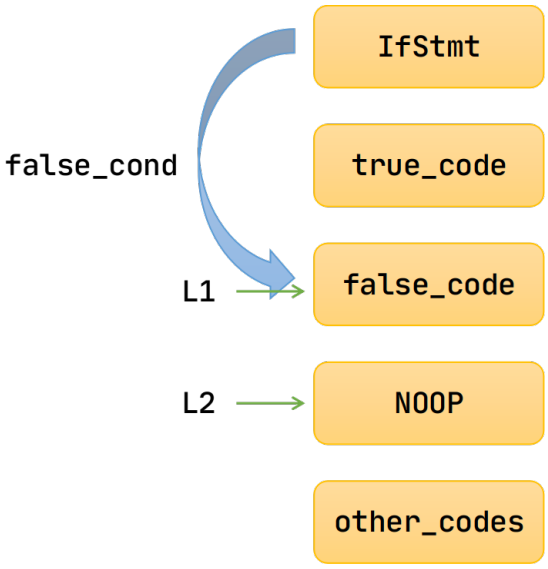
数组采用指针的方式实现，有关数组的操作如下：

- ArrayAllocation：对于申请长度为 L 的数组申请操作，我们使用 piglet 的 HAllocate 指令，分配 $4*(L+1)$ 字节空间，并将首地址返回。
- ArrayAssignment 和 ArrayLookup：首先查找到要赋值或者查找的元素位置，例如如果数组对应的寄存器是 t8，那么 t8 中的值就是数组的首地址，要查找 a[x] 对应的元素，只需要先找到其对应的地址 $t8+(x+1)*4$，再 HLOAD 或 HSTORE 即可。

需要注意的是：我们分配 $4*(L+1)$ 字节的空间，是为了在首位记录数组长度，用作返回数组长度。但随后并未对越界进行判定。

2.3.2 分支、循环语句的处理

- IfStatement：分支语句的设计如图所示：

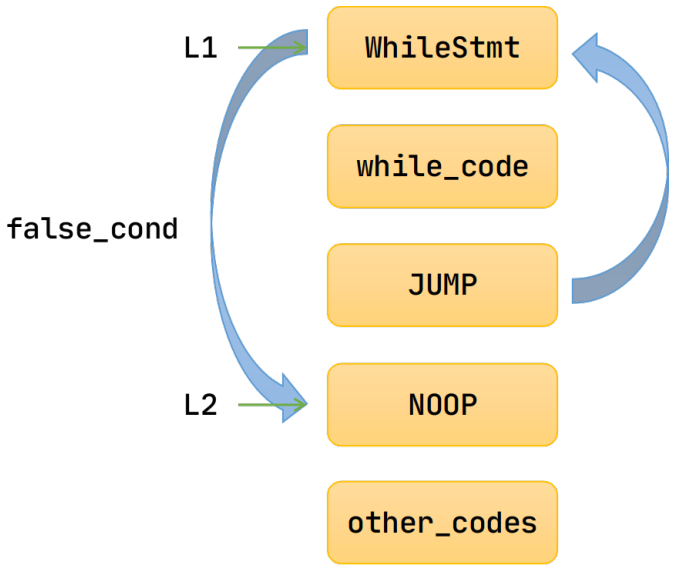


代码部分如下：

```
1     public MType visit(IfStatement n, MType argu) {
2         int L1 = labelNum++;
3         int L2 = labelNum++;
4         PigletPrinter.myPrintWithTab("CJUMP "); // jump when conditions are not satisfied
5         n.f2.accept(this, argu);
6         PigletPrinter.myPrintlnWithTab("L" + L1);
7         n.f4.accept(this, argu);
8         PigletPrinter.myPrintln("");
9         PigletPrinter.myPrintlnWithTab("JUMP L" + L2);
10        PigletPrinter.myPrint("L" + L1); // add a label
```

```
        PigletPrinter.myPrintlnWithTab("NOOP"); // after label, add NOOP to avoid bugs
        n.f6.accept(this, argu);
        PigletPrinter.myPrintln("");
        PigletPrinter.myPrint(String.format("L%d", L2));
        PigletPrinter.myPrintlnWithTab("NOOP");
        return null;
    }
```

- WhileStatement：循环语句的设计如图所示：

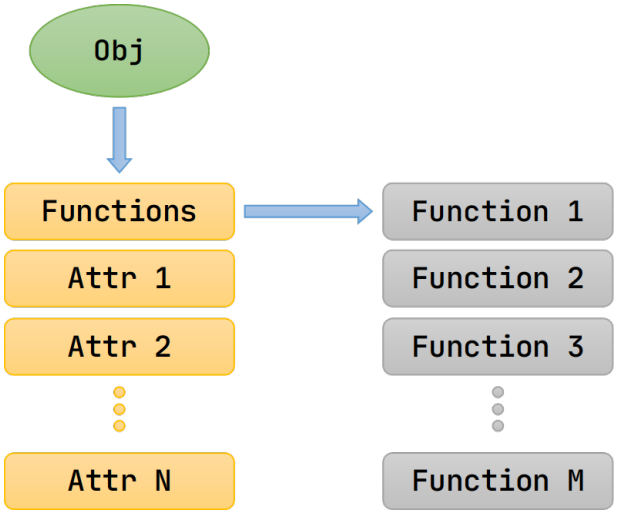


代码部分如下：

```
    public MType visit(WhileStatement n, MType argu) {
        int label1 = labelNum++, label2 = labelNum++;
        PigletPrinter.myPrintln("");
        PigletPrinter.myPrint("L" + label1);
        PigletPrinter.myPrintWithTab("CJUMP ");
        n.f2.accept(this, argu);
        PigletPrinter.myPrintlnWithTab("L" + label2);
        n.f4.accept(this, argu);
        PigletPrinter.myPrintln("");
        PigletPrinter.myPrintlnWithTab("JUMP L" + label1);
        PigletPrinter.myPrint("L" + label2);
        PigletPrinter.myPrintlnWithTab("NOOP");
        return null;
    }
```

2.3.3 类的实现——变量与方法

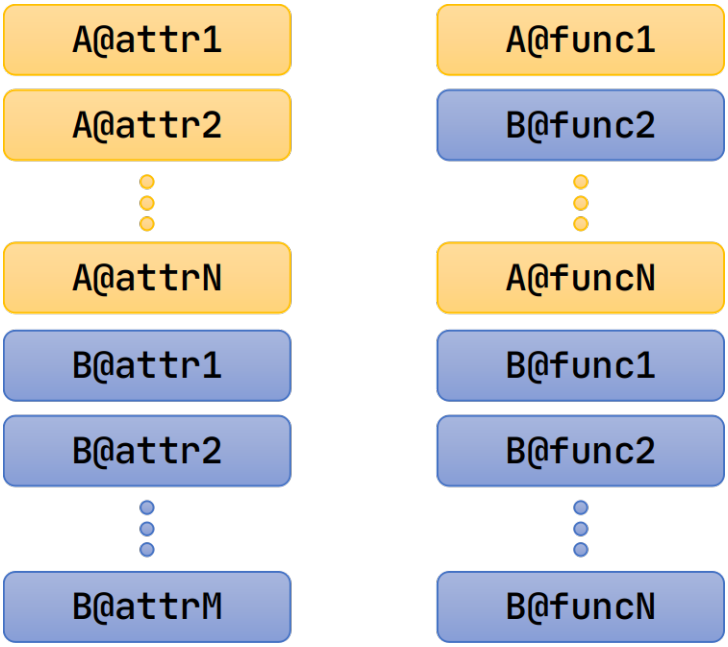
对于每一个对象，我们的处理如图所示：



将对象存储到寄存器时，实际上存储的是对象的首地址。他的前四个字节存储函数表首地址，随后的 N*4 个字节存储其属性。每个类型的变量列表和成员函数列表在进入主程序时就会根据 Lab1 生成的符号表预处理完成。需要注意的是，由于要处理类继承与多态关系等，因此对于每一个类具体的属性与函数列表的预处理放在下节。

2.3.4 多态的处理

预处理时使用 buildVariavles 和 buildMethods 两个函数对每个类构建它的 Variables 和 Methods。此时要考虑到继承关系。方便起见假设 B 继承自 A，那么 B 的成员如图所示：



上图的解释：将当前类的所有父类按顺序从上到下罗列成员变量，变量名为 “class@var”；对于成员函数，如果有同构造的函数（即函数的重写），那么将子类的函数替换掉父类。在构建的同时维护一个变量和函数名字与下标之间的相互映射，便于查找。

首先我们分析这样为什么能完成继承和多态：当分析到 AllocationExpression 时，如果当前的类没有父类，那么按照正常的构建变量的方式即可；如果当前的类有父类，例如上述的 B 类，那么其构建的函数表则如上述右图。此时无论 Allocation 的结果返回给那种类型的变量，在访问涉及多态的函数时，访问的就是其运行时形态。而对于变量的覆盖来说，假设 a 的类型是 A，并且在 AB 两个类都有 x 属性。那么在访问 a.x 时，首先会查找 a 的声明类型为 A，于是就会把要查找（或修改）的变量名定义为 “A@x”，于是就可以查到其对应的位置。

需要注意的是，我们选取 @ 符号作为分隔符的原因是它不会作为变量或者类或者函数名字，但是在构建函数的时候，我们需要构建新的函数名。此时为了避免重名，我们建立了一个从含有 @ 的函数名到不含 @ 函数名的映射，写入在 buildRenamedMap 中。代码如下：

```
1   private void buildRenameMap() {
2       ArrayList<String> allMethods = new ArrayList<>();
3       for (Map.Entry<String, MClass> mClassEntry : MClassList.classList.entrySet()) {
4           MClass curClass = mClassEntry.getValue();
5           for (String method : curClass.methods.keySet()) {
6               String midName = curClass.getName() + "@" + method;
7               String finalName = curClass.getName() + "_" + method;
8               // add '_' to the end until no conflicts
9               while (allMethods.contains(finalName)) {
10                  finalName = finalName + '_';
11              }
12              allMethods.add(finalName);
13              renamedMethods.put(midName, finalName);
14          }
15      }
16  }
```

2.3.5 参数的传递

函数的声明和调用难点都在于参数的存取。对于函数的声明，将参数取出后，对于其余元素直接翻译即可。因此我们重点描述参数的存取。

Piglet 限制参数传递至多 20 个，此时第一个参数（编号 0）需要是 self，1-18 号参数为实参。当参数数量小于等于 18 个时将其存入 1-18，否则将多出的参数存入编号 19 内保存的数组中。编号 19 的寄存器内存入一个地址，这个地址指向存储多余参数数组的首地址。在处理 MessageSend 时的处理路径如下：

3 SPiglet Translate

3.1 SPiglet 介绍

SPiglet 与 Piglet 整体上差异较小，主要差别在于引入了简单表达式，并且不再允许潜逃表达式的出现。本次任务主要是把嵌套表达式分解开。

3.2 SPiglet 翻译

主要过程分两步：

- 第一步：扫描整个程序，获得每个过程的最大寄存器编号 N，新的寄存器从 N+1 开始使用；
- 第二步：对程序进行翻译，主要是 Exp 节点，如果外层是 MOVE 节点那么可以直接继续遍历该节点内部；否则需要先对其求值存储到临时寄存器之后，再返回临时寄存器。代码如下：

```
1     public String visit(Exp n, String argu) {
2         if (outMove) {
3             outMove = false;
4             String retValue = n.f0.accept(this, argu);
5             outMove = true;
6             return retValue;
7         }
8 
9         int curTemp = Counter.MaxTemp.get(argu);
10        Counter.MaxTemp.put(argu, curTemp + 1);
11        SPigletPrinter.myPrintln(String.format("MOVE TEMP %d %s", curTemp,
12                n.f0.accept(this, argu)));
13        return String.format("TEMP %d ", curTemp);
14    }
```

4 Kanga Translate

4.1 Kanga 介绍

在这一部分，我们将把 SPiglet 程序翻译为 Kanga 程序，两者的最主要区别在于 Kanga 的寄存器是有限的，我们需要考虑分配问题。具体而言，Kanga 有如下特点：

- 只有 24 个寄存器。
  - a0-a3 用于存放向子函数传递的参数；

  - v0-v1 用于存放子函数返回结果、表达式求值，从栈中加载；
  - s0-s7 用于存放局部变量，在发生函数调用时一般要保存它们的内容；
  - t0–t9 用于存放临时运算结果，在发生函数调用时不必保存它们的内容。

- 需要管理运行栈。有专门的指令用于从栈中加载（ALOAD）、向栈中存储（ASTORE）值。
- Call 指令无显式参数，使用寄存器和栈存放参数。也没有显式 “RETURN”，结果存储在 v0 寄存器中。
- 过程头部需要包含三个整数，分别指示参数个数、需要的栈单元个数、最大参数数目。

我们的翻译由两遍 Pass 构成，第一遍由 BuildGraphVisitor 为每个方法构建局部控制流图，并最后进行活性分析和寄存器分配。第二遍由 KangaTranslator 进行翻译。

4.2 构建控制流图

我们为每个方法建立一个 MMethod 对象，内部存放一个控制流图，并用 MMethodList 存放所有 MMethod。源程序中的每条语句对应图中的一个顶点，我们用两个 map 分别记录每个顶点对应的出入边。顶点（Vertex 对象）内部记录 gen、kill 等信息用于活性分析。

遍历每个方法时，我们首先为 entry 建立一个结点，再为之后遇到的每条语句建立一个节点，最后为 exit 建立一个结点。由于我们的访问是顺序的，对于除了 JUMP 以外的所有语句，我们都可以添加和下一个结点之间的有向边。对于 CJUMP 和 JUMP 语句，一个可能的问题在于访问到该语句时，跳转的 Label 尚未访问，因此无法得知需要连接哪个结点。为此，我们记录 CJUMP/JUMP 结点到 Label 的映射，并记录 Label 到当前结点（行号）的映射。在完整遍历方法后，添加这些跳转边。

其次，我们需要记录每条语句的 kill/gen 集合。具体而言，我们将 HLoad/Move 的左值加入到 kill 中，将其它所有出现的 TEMP 加入到 gen 中。我们需要为每个 TEMP 变量初始化它的活跃区间，对于 <20 的 TEMP 变量（参数），其初始区间为方法起始到当前行；对于其余 TEMP 变量，其初始区间为当前行到当前行。

在建图过程中，我们额外记录所有的 call site 位置，用于寄存器分配。我们同时更新当前方法调用所需要的最大参数数目，并初始化需要的栈单元个数为溢出的参数个数。

4.3 寄存器分配

首先，我们每个方法进行活跃变量分析，我们采用的方法是流图分析，内部算法为简单的不动点迭代。即不断尝试结点，进行交汇和结点更新运算。如果有变化，则将当前结点和它的所有前驱加入待尝试结点中。

特别地，我们需要移除 exit 结点，避免迭代中额外的判断。由于活跃变量分析是从后向前传播的，我们将初始结点按行号由大到小排序，以加快迭代。

```
    public void buildLiveVars() {
        LinkedList<Vertex> curVars = new LinkedList<>();
        curVars.addAll(graph.vertexMap.values());
        Collections.sort(curVars);
        curVars.removeFirst();
        while(!curVars.isEmpty()) {
            Vertex q = curVars.poll();
            if (q.update(graph)) {
                curVars.add(q);
                for (Integer line: graph.pred.get(q.line)) {
                    Vertex p = graph.vertexMap.get(line);
                    if (!curVars.contains(p))
                        curVars.add(p);
                }
            }
        }
    }
    public boolean update(CGraph graph) {
        HashSet<Integer> savedLive = live;
        live = new HashSet<>();
        for (Integer line: graph.succ.get(line)) {
            live.addAll(graph.vertexMap.get(line).live);
        }
        live.removeAll(kill);
        live.addAll(gen);
        if (live.equals(savedLive))
            return false;
        return true;
    }
```

在获得图中每个结点的活跃变量信息后，我们再次遍历控制流图，更新 TEMP 变量的活性区间。同时，我们利用 call sites 信息，记录这些变量是否会跨过函数调用。

下一步，我们使用线性扫描算法进行寄存器分配。具体而言，我们维护一个“活动”优先队列（activeInterval），记录放在寄存器中的变量，队列中的变量以活性区间结束点为序。我们首先将所有变量存入 intervalList 中，并按照起始点由小到大遍历。当一个新的活性区间起始点到来时，我们顺序扫描活动队列，移除过期变量，并试图将新变量加入队列。当寄存器不足时，我们“溢出”结束最晚的变量。

特别地，如果当前变量不跨过程调用，我们会优先分配 t 寄存器，否则就只考虑 s 寄存器。因此，如果变量跨过程调用，我们溢出的是使用 s 寄存器中结束最晚的变量。

```
    public void linearScan() {
        LinkedList<Interval> intervalList = new LinkedList<>();
        PriorityQueue<Interval> activeInterval = new PriorityQueue<>(Interval::compareByEd);
        intervalList.addAll(intervals.values());
```

```
        intervalList.sort(Interval::compareBySt);
        for (Interval interval: intervalList) {
            Iterator<Interval> iter = activeInterval.iterator();
            while (iter.hasNext()) {
                Interval q = iter.next();
                if (q.ed < interval.st) {
                    freeReg(q);
                    iter.remove();
                }
            }
            if (checkReg(interval))
                activeInterval.add(interval);
            else
                spillReg(activeInterval, interval);
            sRegNumMax = Long.max(sRegNumMax,
                Arrays.stream(sRegUsed).filter(x -> x==true).count());
        }
        stackNum += sRegNumMax;
    }
```

4.4 翻译细节

在完成寄存器分配后，翻译工作整体比较简单。主要工作在于为遇到的每个 SPiglet 里的 TEMP 变量找到对应的存储位置（寄存器或栈）。我们为此提供两个函数作为接口。

```
    public String getReg(String destReg, int num, MMethod curMethod, boolean inKill) {
        if (curMethod.tReg.containsKey(num))
            return "t" + curMethod.tReg.get(num);
        else if (curMethod.sReg.containsKey(num))
            return "s" + curMethod.sReg.get(num);
        else if (!inKill){
            KangaPrinter.myPrintln("ALOAD " + destReg
                + " SPILLEDARG " + curMethod.stack.get(num));
        }
        return destReg;
    }

    public void saveReg(String srcReg, int num, MMethod curMethod) {
        if (curMethod.tReg.containsKey(num) || curMethod.sReg.containsKey(num))
            return;
        else
            KangaPrinter.myPrintln("ASTORE SPILLEDARG " + curMethod.stack.get(num)
                + " " + srcReg);
    }
```

如果 TEMP 变量值存在寄存器中，我们直接返回对应寄存器即可。如果存在栈中，我们需

要考虑为此变量是否被赋值。如果不是，我们用 ALOAD 指令加载变量到临时寄存器（v0/v1）；否则，我们只需返回临时变量，用以存放计算结果。对于栈中的变量，如果它被更新，我们还需要用 ASTORE 指令将其存回栈中。

特别地，我们需要仔细考虑存储临时变量的寄存器（v0/v1）的使用，避免出错。考察 Kanga 语言，同一条语句最多只会使用两个临时寄存器的值，因此无需考虑冲突的问题。此外，在翻译 CALL 语句时，需要首先加载参数到寄存器和栈中。而在翻译方法时，起始需要将可能覆盖的 s 寄存器保存到栈中，并将参数加载到我们上一轮为其分配的 s/t 寄存器或栈中；而在返回时，需要将返回值加载到 v0 寄存器中，并恢复 s 寄存器的值。

5 MIPS Translate

5.1 任务简介

本次任务要求将上一部的 Kanga 代码翻译为 MIPS 代码，主要任务在于将之前的入栈出栈手动操作，自己实现栈空间的分配。同时，对于一些允许操作立即数和标签的 Kanga 语法，在 MIPS 中要先将其存入寄存器中并对寄存器操作。由于本次 Lab 并不涉及复杂的 MIPS，主要难点在于栈的实现上，因此我们并没有深入了解 MIPS，只是了解了我们需要用到的指令。本次 Lab 所有代码均位于 kanga.visitor.MIPSTranslator 内。同时，因此本次代码缩进比较简单，因此不在单独设置 Printer 类，而是替换为 printWithTab 和 printNoTab 两个函数。

5.2 翻译细节描述

5.2.1 函数的翻译

每个函数开始都需要：

- 保存帧指针 $fp 和返回地址 $ra；
- 分配新的栈空间，将栈指针下移，栈空间分配见下一节；

当函数结束时需要：

- 加载帧指针和返回地址；
- 将栈指针上移；

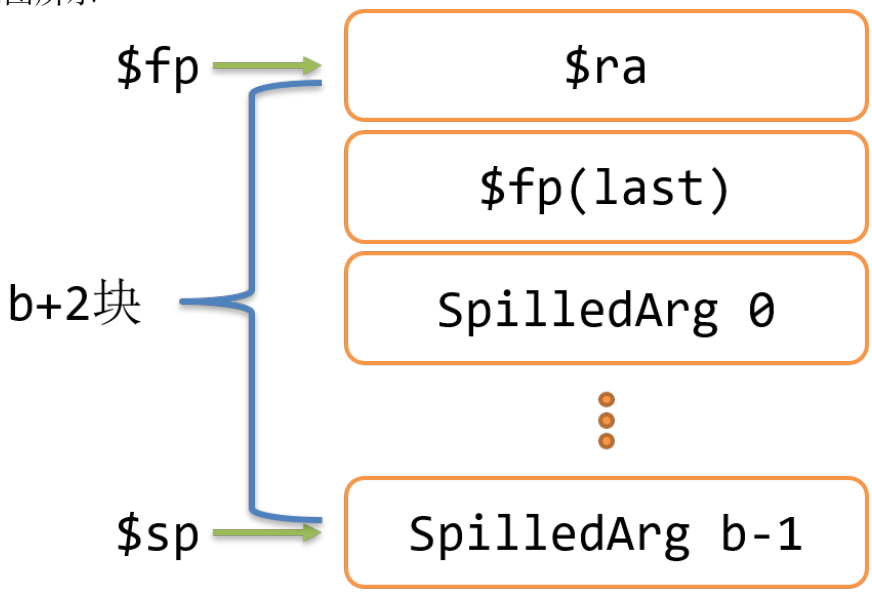
因此，处理过程调用的代码如下：

```
1    public String visit(Procedure n, String argu) {
2        /*code for calculating stack_size*/
3
4        printWithTab(".text");
```

```
5           printWithTab(String.format(".globl %s", func_name));
6           printNoTab(func_name + ":\n");
7           printWithTab("sw $fp, -8($sp)");
8           printWithTab("sw $ra, -4($sp)");
9           printWithTab("move $fp, $sp");
10          printWithTab(String.format("subu $sp, $sp, %d", stack_size));
11          n.f10.accept(this, argu);
12          printWithTab(String.format("addu $sp, $sp, %d", stack_size));
13          printWithTab("lw $fp, -8($sp)");
14          printWithTab("lw $ra, -4($sp)");
15          printWithTab("j $ra\n");
16          return null;
17      }
```

5.2.2 栈的实现

如图所示：



栈的结构为 main 函数分配 (b+1)*4 的空间，其余每个函数分配 (b+2)*4 大小的栈空间，函数的参数记录在当前函数的栈空间内。开头的四个字节为返回值寄存器，第二个四字节为上一次栈帧寄存器。同时，栈内元素的标号与栈地址高低相反。因为每个函数的参数在自己当前栈内，所以在上一帧调用当前函数时，应已经将参数传入，因此应当将标号较小的栈位置（溢出参数）放在高地址，标号较大的栈位置（函数内的溢出）放在低地址。

5.2.3 其它细节

- MOVE 语句的翻译：由于 Move 指令是唯一允许右值为 Exp 的语句，因此它的处理相对复杂一些。与此同时，Binop 表达式也只会在此出现，因此将 Binop 的处理也移交至此。主要有以下几种情况：
  - 右值为 HAllocate：这时需要进行系统调用，其返回值位于 $v0，首先依照之后介

绍的系统调用的方式获得返回值后，使用 move 指令将 $v0 赋值给左值即可；

- 右值为 SimpleExp
  - 为寄存器：直接使用 MIPS 指令中的 move；
  - 为立即数：直接使用 MIPS 指令中的 li；
  - 为标签：直接使用 MIPS 指令中的 la；
- 右值为 BinOp：此时右值为 SimpleExp，仍然需要分情况讨论右值：
  - 为标签或立即数：将其存入 $v0 或 $v1 两个临时寄存器之一（不与左值冲突即可）；
  - 为寄存器：可以直接进行运算，无需额外操作

- PASSARG 与 SPILLEDARG：这两个操作为 Kanga 中对栈进行的操作，比较友好的是，在 Kanga 中这些操作已经给出了地址，因此我们只需要根据给出的地址在当前函数的溢出空间寻址即可；
- 系统调用：系统调用则遵循课件中讲的，将调用号存入 $v0 中，调用参数存入 $a0 中，并从 $v0 中获取内存申请的返回值。由于本次 Lab 不需要其余操作，因此并未了解其余 MIPS 调用号的用处。

6 Summary

- 我们的代码链接：`https://github.com/malusamayo/MinijavaLab`；
- 其中，我们为每一个 lab 编写了测试脚本，首先将 java 文件编译至 bin/目录下后，运行”python3 run_lab*_test.py” 即可，对于最后一个 lab 需要事先在 Linux 下编译安装 SPIM 的 CLI 版本环境（Windows 无 CLI 版本）；
- 感想与体会：编译实习课程上，我们真正实践到了编译理论课程上学到的算法与知识。对如何制造一个编译器有了更深刻的理解。我们两个本学期都是编译实习与编译技术两门课同步选的，很多时候实践可能比理论进度还快，在学到理论课的时候感觉理解起来也更加轻松了。