minijava2piglet

将符合minijava规范的代码翻译成符合piglet规范的代码。

***symbol.\****

在第一阶段类型检查基础上添加的内容有：

MClass：

1. field varNo：field在当前class的序号，初始1，有getVarNo方法
2. field methodNo：method在当前class的序号，初始0 ，有getMethodNo方法

MMethod：

1）field paramTempNo：分配参数的TEMP号码，初始1，在本method中增长，有getParamTempNo方法

2）field NoinClass：表示当前method是在所属类的第几个method，初始0

MVar：

1）field classOffset：class的第n个field，存在TEMP0偏移n\*4处

2）field methodTempNo：在method中的TEMP号码，局部变量存在TEMP 20及以上，参数存在TEMP 1-19

构造函数增加两个参数分别用来设置classOffset和methodTempNo，用不到的另一个field置为-1

***BuildSymbolTableVisitor.java***

在第一阶段类型检查基础上修改的内容有：

VarDeclaration:

((MClass)argu).insertVar(new MVar(t.getType(), id.getName(), ((MClass)argu).getVarNo()\*4, -1));

((MMethod)argu).insertVar(new MVar(t.getType(), id.getName(), -1, ((MMethod)argu).getLocalTempNo()));  //getLocalTempNo()通过Temp.java中的getTempNo20分配TEMP 20以上的TEMP号码

MethodDeclaration:

new MMethod(returnType.getType(), id.getName(), ((MClass)argu).getName(), ((MClass)argu).getMethodNo());//最后一个参数设置NoinClass

FormalParameter:

((MMethod) argu).addParam(new MVar(t.getType(), id.getName(), -1, ((MMethod)argu).getParamTempNo()));

***Temp.java***

getTempNo20：分配TEMP 20以上的TEMP号码，初始20，全局增长。

***PrinterPg.java***

记录翻译结果

***PigletVisitor.java***

**AllocationExpression**

BEGIN和END之间，构建实例表

分配Dtable: TEMP i 存放Dtable起始地址

MOVE TEMP i HALLOCATE "4\*method个数" （method个数直接通过GlobalData.allClassList符号表中要new的目标class有多少个method得到）

HSTORE TEMP i "4\*NoinClass" "className\_methodName" (有多少个method就有多少行，NoinClass是MMethod的field，记录当前method是在所属类的第几个method，初始值为0)

分配Vtable: TEMP i+1 存放Vtable起始地址，用TEMP i+1 是因为新分配TEMP序号递增

MOVE TEMP i+1 HALLOCATE "4\*(field个数+1)" （field个数直接通过GlobalData.allClassList符号表中要new的目标class有多少个field得到）

HSTORE TEMP i+1 4 0

HSTORE TEMP i+1 8 0

...(有多少个field就有多少行，假设有a个，所有field初始化为0)

HSTORE TEMP i+1 4\*a 0

HSTORE TEMP i+1 0 TEMP i (Vtable的第一块存放Dtable的地址)

RETURN TEMP i+1

**MessageSend**：PrimaryExpression.Identifier(...)

CALL

BEGIN

MOVE TEMP i <PrimaryExpression子树>

PrimaryExpression有四种可能，归根结底都是某个对象，需要知道该对象是哪个class的实例

1) ThisExpression

visit的第二个参数argu传递过来的是当前method的符号表即MMethod类(MessageSend总是在method中)，MMethod类会记录它所属的MClass

2）Identifier

通过n.f0.accept(this, argu)的返回值得到className(f0是PrimaryExpression)

但因为可能存在上溯造型的情况，所以在argu传递的符号表/MMethod中根据PrimaryExpression作为Identifier的name查找到本Identifier，得到它的运行时类型

3）BracketExpression

通过n.f0.accept(this, argu)的返回值得到className(f0是PrimaryExpression)

4) AllocationExpression

通过n.f0.accept(this, argu)的返回值得到className(f0是PrimaryExpression)

Identifier是method name，有同名方法的话执行运行时类型的方法，总之是尽可能执行子类的方法，子类里找不到才往父类寻找该方法。最后找到要调用的method真正属于的class。

HLOAD TEMP i+1 TEMP i 0 （TEMP i 指向PrimaryExpression那个对象的Vtable，这一行是在把Vtable的0偏移处的Dtable地址加载到TEMP i+1）

HLOAD TEMP i+2 TEMP i+1 "offset" (向TEMP i+2 加载Identifier所表示的method在实例表中的地址，TEMP i+1存的是Dtable的地址，偏移量offset是method在Dtable中的偏移量)

RETURN TEMP i+2

END (TEMP i <ExpressionList子树>) 【第一个参数是对象的实例表Vtable地址】

**AssignmentStatement**：Identifier=Expression;

visit的第二个参数argu传递过来的是当前method的符号表即MMethod类，也可以由当前method获取所属class的信息（MClass/符号表）

判断Identifier是当前method的临时变量还是函数参数，还是当前class的field

从符号表中查找得到Identifier对应的MVar后，MVar中的classOffset （class的第n个field，存在TEMP0偏移n\*4处）和methodTempNo（在method中的TEMP号码，局部变量存在TEMP 20及以上，参数存在TEMP 1-19）这两个field给出翻译需要的信息。

1）method的临时变量或函数参数

MOVE TEMP x <Expression子树> 【TEMP x就是methodTempNo记录的TEMP号码】

2）class的field

HSTORE TEMP 0 "classOffset" <Expression子树> 【TEMP 0 是当前对象实例表Vtable的地址，根据偏移量找到目标field的地址，将Expression子树得到的存储到目标field的位置，实现赋值】

这部分还可能涉及上溯造型的情况，我写的Expression accept之后会返回类型名称，如果返回的类型名称和Identifier的类型不同，要修改Identifier这个MVar的运行时类型信息，在GlobalData.allClassList符号表中进行修改。

**ArrayAssignmentStatement**：Identifier[Expression1]=Expression2;

Identifer可能的情况：

1）当前method的临时变量或函数参数

从符号表里得到相应的MVar，由其中的methodTempNo得到相应的TEMP x

MOVE TEMP i TEMP x

2) 当前class的field

MOVE TEMP i

BEGIN

HLOAD TEMP i+1 TEMP 0 "classOffset"

RETURN TEMP i+1

END

TEMP 0 是当前对象实例表Vtable的地址，根据偏移量找到目标field的地址，从中加载目标field的值到TEMP i+1，再到TEMP i

MOVE TEMP i+2 <Expression1子树>

HLOAD TEMP i+3 TEMP i 0 【TEMP i 存储的应该是Identifer数组的首地址，首地址偏移0处存储数组长度，存储到TEMP i+3】

CJUMP LT TEMP i+2 TEMP i+3 Lj 【比较Expression1得到的索引值和数组长度，如果数组越界跳转到Lj ERROR】

JUMP Lj+1 【没有数组越界的问题的话跳转到Lj+1(用j,j+1来表示lable号码只是因为我是号码递增这样分配的)】

Lj NOOP

ERROR

Lj+1 NOOP

MOVE TEMP i PLUS TEMP i TIMES 4 PLUS 1 TEMP i+2 【计算出Identifier[Expression1]的地址，索引值要先加1是因为0偏移处存放数组长度了】

HSTORE TEMP i 0

<Expression2子树>

**ArrayLookup**：PE1[PE2] (PE即PrimaryExpression）

BEGIN

MOVE TEMP i <PE1> 【数组基地址】

MOVE TEMP i+1 <PE2> 【数组索引】

HLOAD TEMP i+2 TEMP i 0 【加载数组长度到TEMP i+2】

CJUMP LT TEMP i+1 TEMP i+2 Lj 【检查数组越界】

JUMP Lj+1

Lj NOOP

ERROR

Lj+1 NOOP

MOVE TEMP i PLUS TEMP i TIMES 4 PLUS 1 TEMP i+1 【计算基地址+偏移量】

HLOAD TEMP i+3 TEMP i 0 【从基地址+偏移量处加载内容】

RETURN TEMP i+3

END

**ArrayLength**：从数组基地址0偏移量处获得数组长度

**ArrayAllocationExpression**:new int[Expression]

BEGIN

MOVE TEMP i <Expression>

MOVE TEMP i+1 HALLOCATE TIMES 4 PLUS TEMP i 1 【计算得到要为该数组分配的长度，为4\*(Expression值+1)，+1是因为0偏移处还要存数组长度；分配相应长度的地址空间，返回基地址存储给TEMP i+1】

HSTORE TEMP i+1 0 TEMP i 【把数组长度存储到数组基地址0偏移处】

MOVE TEMP i+2 0 【TEMP i+2 从0到TEMP i存的数组长度-1 遍历，实现数组初始化置0】

Lj CJUMP LT TEMP i+2 TEMP i Lj+1

MOVE TEMP i+3 PLUS TEMP i+1 TIMES 4 PLUS 1 TEMP i+2

HSTORE TEMP i+3 0 0

MOVE TEMP i+2 PLUS TEMP i+2 1

JUMP Lj

Lj+1 NOOP

RETURN TEMP i+1 【返回数组基址】

END

**Identifier**：返回类型名称

1) class名：无翻译，直接返回类型名即可

2）当前method的临时变量或函数参数：visit的第二个参数argu传递过来的是当前method的符号表即MMethod类，向其查询得到MVar，从中得到TEMP号码x，翻译TEMP x

3）当前class的field：要考虑继承的情况，如果当前class没有这个名称的field，向父类查找。TEMP 0存放当前实例表Vtable基地址，目标field在表中的偏移量向符号表查询得到，从Vtable基地址+偏移量处取值。

4）当前class的method：这种情况只会出现在MessageSend中，而MessageSend中直接用tokenImage了，没有进入Identifier的visit()

**IfStatement**：if(Expression)Stmt1 else Stmt2

MOVE TEMP i <Expression子树>

CJUMP TEMP i Lj 【Expression为真，执行下一条指令，即Stmt1的翻译结果；Expression为假，跳转到Lj执行Stmt2的翻译结果】

<Stmt1>

JUMP Lj+1 【执行完Stmt1的翻译结果跳到这段最后】

Lj NOOP

<Stmt2>

Lj+1 NOOP

**WhileStatement**：while(Expression) Stmt

Lj MOVE TEMP i <Expression>

CJUMP TEMP i Lj+2 【Expression为真，执行下一条指令，即Stmt的翻译结果；Expression为假，跳转到Lj+2结束这段】

Lj+1 NOOP

<Stmt>

JUMP Lj 【执行完Stmt的翻译结果跳转到Lj再次进行条件判断】

Lj+2 NOOP

piglet2spiglet

将符合piglet规范的代码翻译成符合spiglet规范的代码，主要工作需要消除掉嵌套表达式，除了move可以使用表达式，print可以使用简单表达式外，其他语句中都需要消除表达式使用临时变量TEMP。因此符合spiglet规范的代码比符合piglet规范的代码要多使用很多TEMP。

***getTempVisitor.java***

先遍历一遍语法树，找到符合piglet规范的代码中用过的最大的TEMP值，以便翻译成spiglet的时候分配没分配过的TEMP。spiglet规范下临时变量TEMP 可使用的个数无限制。

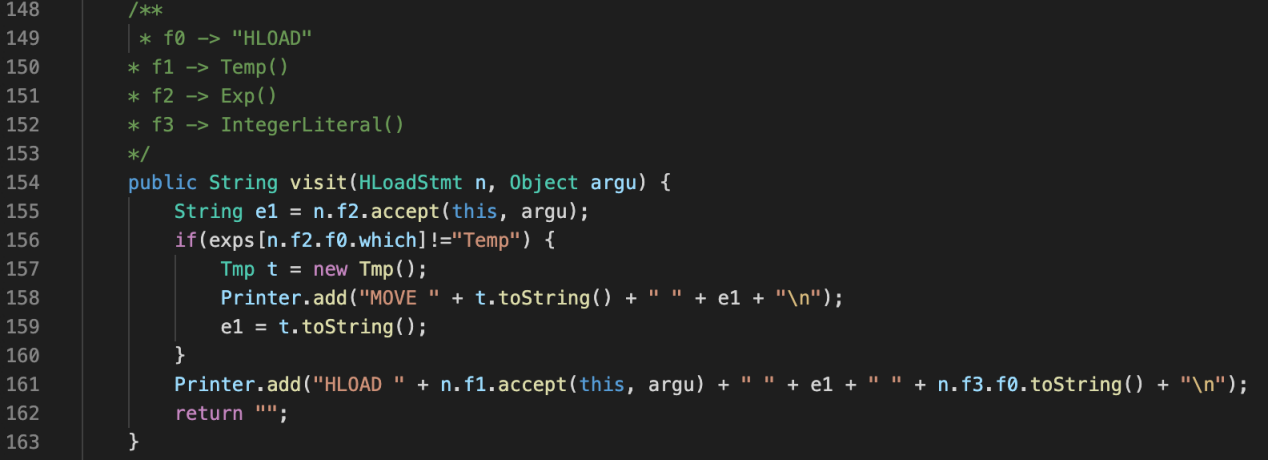
***Tmp.java***

管理临时变量TEMP，static field cnt先通过getTempVisitor中对setInitCnt的调用设置成大于等于20的符合piglet规范的代码中用过的最大的TEMP号码，在后续翻译过程中需要使用临时变量时就构造一个Tmp，field num表示TEMP号码，cnt全局增长的同时给num赋值进行TEMP号码分配。

***SpigletVisitor.java***

实现从符合piglet规范的代码到符合spiglet规范的代码的翻译，在每个节点使用String类型的返回值，不使用参数，需要时用返回值传递父节点需要使用的子节点的翻译内容。访问Stmt节点时直接进入NoOpStmt、HStoreStmt等子节点分别处理翻译，访问Exp节点时也同样直接进入StmtExp、Call等子节点分别处理翻译，翻译结果保存在Printer的static field out中。

以HLoadStmt为例：



155行等号右边进入Exp节点完成了Exp节点的翻译，翻译结果不涉及HLoadStmt节点翻译的部分直接保存到Printer中，HLoadStmt节点需要使用的部分返回赋值给e1。

156行判断f2这一Exp节点的类型，如果不是Temp节点，需要有将e1结果move到一个临时变量的步骤，更新e1为新分配的临时变量的String形式即”TEMP x”。

161行完成无表达式只有临时变量的HLOAD语句翻译。

spiglet2kanga

将符合spiglet规范的代码翻译成符合kanga规范的代码。kanga和spiglet的主要区别有kanga只有有限的寄存器，开始使用运行栈，Call语句过程调用处也有差别。

实现spiglet2kanga分为三个步骤：活性分析、寄存器分配和代码翻译。

活性分析使用流图分析法，具体代码由退课的队友实现。

***RegisterAllocation.java***

寄存器分配使用线性扫描算法。由活性分析可以获得各个变量的活性区间，保存为GlobalData中以String形式的TEMP编号（每个procedure独立的TEMP 0-19 表示成”procedureName\_TEMP号码”的形式）为key，VarInterval类型（field包括临时变量名、全局下该临时变量活性区间的起始和结束行数，行数只计算有临时变量的行）为value的HashMap varinfo。

首先将各个procedure的函数参数加入varInfo，并且建立一个按照活性区间起始位置排序的VarInterval的TreeSet，声明为all，将varInfo中的value集中的VarInterval都加入all。

寄存器分配的主要过程在registerAllocation中实现，目标是在每个有活性区间起始的行数处都构造一张分配表，用AllocationTable类型实现，AllocationTable中包含所在行数pos、寄存器分配情况regAlloTable、运行栈情况spill2Stack等信息。GlobalData中以行数为key、AllocationTable为value的raInfo记录所有分配表，并按照行数升序排序。

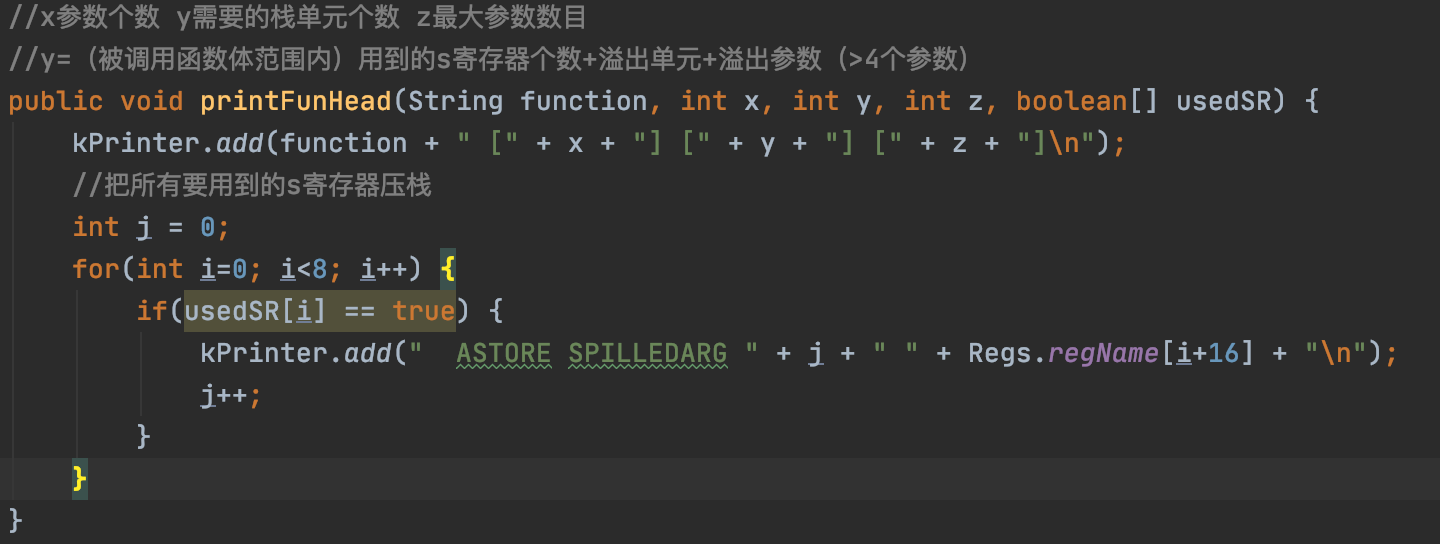
维护一个active队列，记录放在寄存器中的变量，使用TreeSet，按活性区间结束处排序。

寄存器分配过程：遍历排序后的活性区间，遇到一个新的活性区间，先把raInfo中前一个AllocationTable深拷贝下来，如果前一个分配表的位置和现在这个活性区间的起始位置一样的话，就把前一个分配表从raInfo删除。顺序扫描active队列，移除过期变量，试图将新变量加入队列：如果active队列的大小小于通用寄存器总数，则可以分配寄存器给当前变量，Register.java负责寄存器情况的记录和寄存器的分配；否则是寄存器不足的情况，需要溢出活性区间结束最晚的变量到运行栈。

***toKangaVisitor.java***

实现从符合spiglet规范的代码到符合kanga规范的代码的翻译，翻译过程与piglet2spiglet类似。访问Stmt节点时直接进入NoOpStmt、HStoreStmt等子节点分别处理翻译，访问Exp节点时也同样直接进入HAllocate、Call等子节点分别处理翻译，翻译结果保存在kPrinter的static field out中。

访问Goal节点和Procedure节点时，需要调用printFunHead函数和printFunTail函数完成一个过程首尾的例行工作翻译。



对Temp节点的处理调用readFromTemp函数和writeToTemp函数实现，前者在遇到要从该变量中读内容的情况时使用，后者在遇到要向该变量写内容的情况时使用。raInfo在此用到，根据当前目标spiglet代码的行数向raInfo从当前行数往前寻找最近的一张分配表，根据分配表的寄存器分配和运行栈情况进行相应的翻译。

kanga2mips

将符合kanga规范的代码翻译成符合mips规范的代码，mips是一种汇编代码，需要包含汇编信息，需要根据mips的寄存器约定翻译，mips指令虽然看起来和之前几个阶段有很大不同，但实际翻译起来和kanga挺一一对应的，主要比kanga多了栈结构的维护部分。

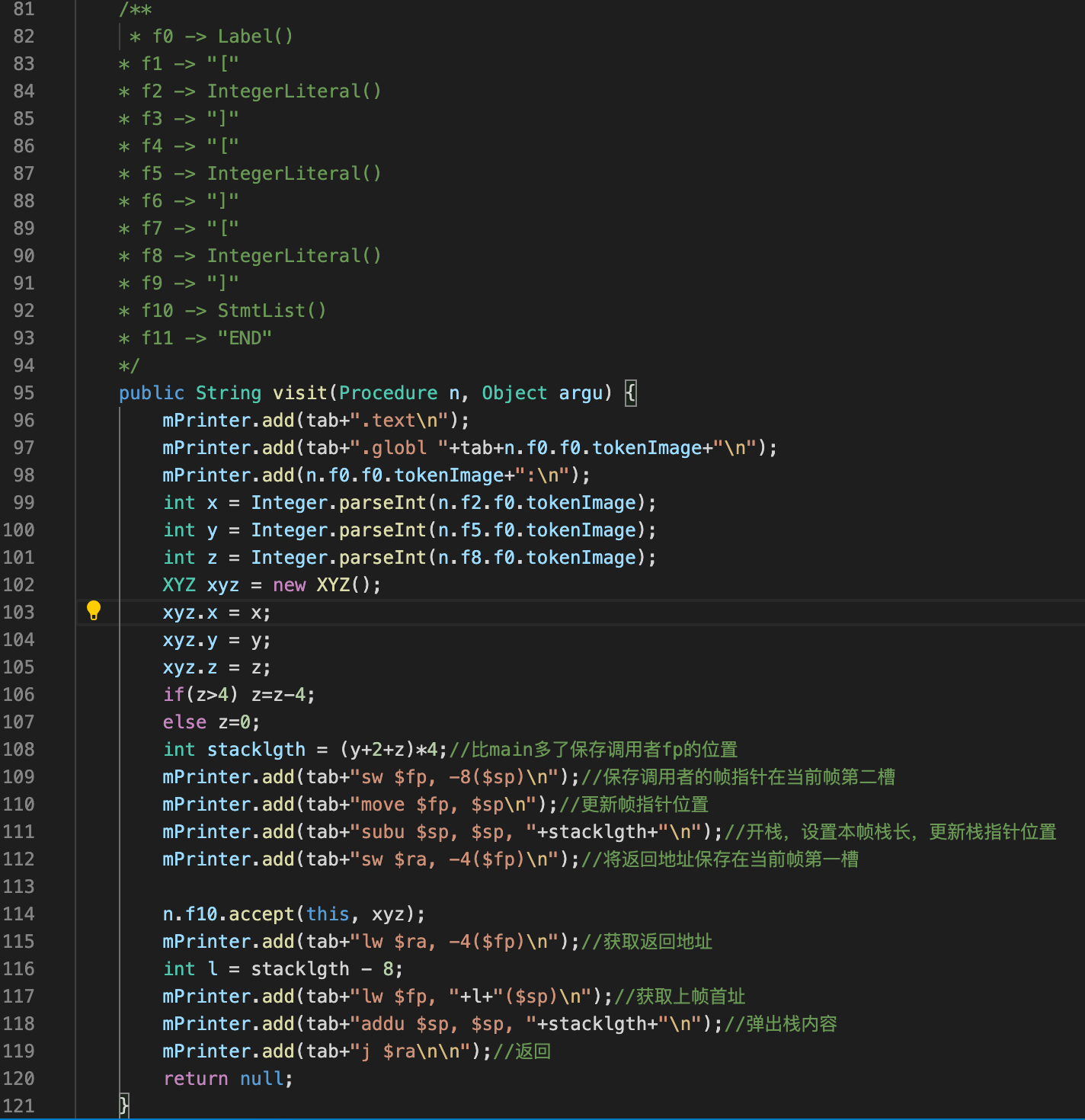
在每个procedure（包括main）的起始都需要开栈，栈长的计算方式如下：

procedureName [x] [y] [z]

zz = (z>4)?(z-4):0;

Main的栈长为4\*(y+zz+1)：y是保存局部变量的，z大于4的部分用于传参，1是放返回地址的。

其他procedure的栈长为4\*(y+zz+2)：比main多了保存调用者fp的位置。



XYZ类型用来在语法树的节点之间传递当前procedure的参数信息。

mips的指令集非常丰富，需要根据Exp、SimpleExp的类型决定指令，这一点在MoveStmt节点的翻译中充分体现了：



