Group 11, Exercise 4

Madhusudhan Subramanya, msubrama@rhrk.uni-kl.de,
Rohan Shekar Thingalaya, thingala@rhrk.uni-kl.de,
Rama Klaho, klaho@rhrk.uni-kl.de,
Monireh Pourjafarian, mpourjaf@rhrk.uni-kl.de

19 June 2017

1 Translation of Minijava statements to LLVM

We have written the Translator class inside the Translator.java file according to test cases. As a general approach, we have Override the cases by visitor pattern (extends Element.DefaultVisitor) and using Matcher to recognize the case easier. We have a main block called **entry** that every LLVM translation will add to it while Matcher execution is running. After it is finished, the entry will add to program block **Proc**. Please find the details of specific statements which has described in the following.

1.1 case StmtIf

The Branch condition, trueLabel, falseLabel are obtained from the MiniJava AST and necessary blocks are created for each labels. An exit block "b3" is created and the true and false Blocks are directed to "b3" and "b3" is directed to the end block.

1.2 case StmtWhile

The exitCondition, loopBody are obtained from the MiniJava AST. A loop block is created called "L0" and the entry block is redirected to Jump to "L0". The L0 contains the corresponding LLVM code of loopBody. The exitCondition is translated to LLVM branch syntax and the control either jumps to "L0" or the exit block "L2". "L2" in turn jumps to the end block later.

1.3 case StmtReturn

In case of Return statement encounter, the current block in the corresponding LLVM code is added with the Operand Instance of the return expression or with the Integer instance of the return expression.

1.4 case StmtPrint

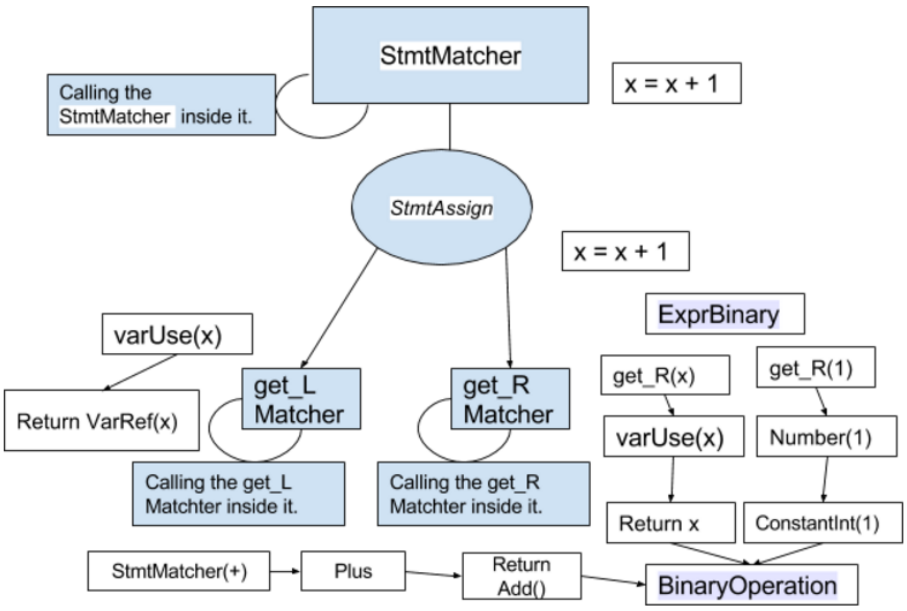
The print Statement obtained from the MiniJava AST. If the statement is of type Operand, a corresponding "Print" equivalent of LLVM is included or the equivalent Integer value is included.

1.5 case StmtExpr

The Expressions are matched for the type Unary or Binary.

1.6 case StmtAssign

For this case, by running the test case, stmtMatcher will identify the statements and run out the related case which is "case-StmtAssign". Our method is depicted in the below, so it should be clear to understand.



For the other statements the approach is some how the same.

1.7 case ExprBinary

The Right and Left Operands are obtained from get-L() and get-R() methods and the corresponding operator is obtained too from the MiniJava AST. A special case of DivideByZero is handled. A new block "L1" is created that includes the BinaryOperation segment and the control is made to jump to the end block later.

1.8 case ExprUnary

The Unary Operand and operator is obtained from the MiniJava AST and a corresponding BinaryOperation with First Operand as zero is inserted in the current LLVM block.

1.9 case BoolConst

A LLVM Boolean constant of the equivalent MiniJava boolean value is returned to the caller.

1.10 case VarDecl and case VarUse

Before using variables we have to make sure that they has declared first. So we have to have a Varblock. For this reason we define a Hash map called **tempVars**. By calling VarDecl we *put* the variable declaration to it and before using it in VarUse first we are checking if the variable declaration is exist in this Varblock then return VarRef of it.

1.11 case Number

A LLVM Constant Integer Value of the equivalent MiniJava number is returned to the caller.

1.12 case ArrayLookUp

For ArrayLookUp we followed the Array access part in the *Translation to Intermediate Representation* script. So we calculated the address, then to prevent memory corruption and information leaks, as it was necessary, we performed a number of checks before loading or modifying the content of the address. For this case, we have three blocks, **L0** which called currentBlock in the code, **L1** which get the value of the a[i], **L2** is the container of Error block, **L3** is the end block.

1.13 case ExprNull

A Temporary Variable is created and a corresponding memory is allocated in the heap with the type TypeNullPointer().

1.14 case NewIntArray

The length of the array is obtained and is multiplied with 4. The result is added with 4. The above two operations are done using the call "BinaryOperation" of LLVM AST. The memory is allocated by taking the resulting value from the above calculation. The size of the array is stored at the top of the stack of the array.