**Programming Assignment 4: IR Generation**

Liyuan Gao (ly342)

Daniel Chen (dlc287)

Darbin Reyes (der34)

*\*This assignment did not require any interfaces from edu.cornell.cs.cs4120.testing to be implemented.*

**Summary**

The purpose of this assignment was to implement a tree based IR (*intermediate representation*) that would bridge the gap between high level source and assembly code. IR generation makes use of the abstract syntax tree (AST) from the previous assignment. Basically, the AST would be traversed and the appropriate translation is generated for each node in the AST.

To implement the IR translation, we created an abstract IRTranslation.java class that distinguishes various types of translations. These translation types are expressions (IRTranslationExpr.java – represented by E in the translation rules), statements (IRTranslationStmt.java – S), and conditions (IRTranslationCondition.java – C). We also implemented a class called IRContext.java and IRContextStack.java. Specifically, the IRContextStack is an array list of the IRContext objects. These classes are used to keep track of registers related to variables, names of the function labels, and arguments during function calls.

A Label.java class was also implemented to represent the labels used in the assembly code. These labels are used as place makers for jump statements. The labels are used for functions and for conditional statements.

The Register.java class keeps track of all the registers used for IR translation. For this translation, it was assumed that there are an infinite number of translations. Within the Register.java class, we also implemented special registers (null, fp – frame pointer, rv – return value) and 6 “free” registers (rdi, rsi, rdx, rcx, r8, and r9). These free registers are used for passing arguments and return values for functions. If the number of arguments exceeds the 6 free registers, then the remaining will be placed on the stack. For a function returning only one value, the rv register is used. If the number of return values exceeds the 6, the first return will be placed in the rv register, the next 5 will be placed in the free registers, and the remaining placed sequentially on the heap. The register r9 will be a pointer to the heap indicating the location of the remaining values.

Our implementation of IR nodes can be found in the xi.ir package. Each node follows the specification described in the Appel book on pages 138~139. The only difference from the spec is that the SEQ node is an array list of statements, rather than 2 statements.

For each AST node, we implemented a method called to\_ir(IRContextStack), which returns an IRTranslation object. An AST node will either return an IRTranslateExpr, IRTranslateStmt, or IRTranslateCondition object. For each respective object, the expr(), stmt(), and cond() methods in the object will return the correct translation.

**Specification**

For short-circuiting AND and OR conditions, we implemented IRTranslateAndCondition.java and IRTranslateORCondition.java classes. These objects are used only in the BoolOpNode.java class, to represent expressions that have an AND or OR operation. The purpose for these additional classes is to distinguish when these operations are used for expressions or conditional jumps. For example, these two cases have different translations when using the AND condition:

|  |  |
| --- | --- |
| if (TRUE & FALSE) | TRUE & FALSE |
| SEQ( C[TRUE,a,f],  LABEL(a),  C[FALSE, t, f],  LABEL(t),  LABEL(f) ) | ESEQ( SEQ( MOVE(TEMP(r), CONST(1)),  C[TRUE & FALSE, t, f],  LABEL(f),  MOVE(TEMP(r), CONST(0)),  LABEL(t) )) |

C[TRUE, l1­, l2] is equivalent to calling:

new IRTranslationExpr(new const(1)).cond().

C[TRUE & FALSE, l1­, l2] is equivalent to calling:

new IRTRanslationAndCondition(new BinOp(BinOp.AND,

new const(1), new const(0)).cond().

For the UnNegNode.java AST node, this is represented in the IR translation has zero minus the expression.

For the UnNotNode.java AST node, the IR translation is an XOR of the expression with the constant 1.

We implemented constant folding at the AST and the IR level. The purpose of implementing constant folding at the AST is to constant fold a larger chunk of the tree. This allows the fewer constant folding operations at the IR level. NEED TO ADD MORE INFO ABOUT IR LEVEL CONST FOLD

**Testing**