CS 536 — Spring 2016

CSX Code Generation Routines

Part II

I/O Statements

All CSX I/O is done by calling members of the class `CSXLib`. To read an integer we call `readInt()`; to read a character we call `readChar()`. To write an integer we call `printInt(int)`; to write a boolean we call `printBool(boolean)`; to write a character we use `printChar(char)`; to write a string we call `printString(String)`; to write a character array we call `printCharArray(char[])`.

We start with read statements. A read is translated like an assignment, where the source of the assignment is produced by a call to `readInt` or `readChar`.

```
void genCall(String methodDescriptor){
// Generate a static method call:
//    invokestatic methodDescriptor
}

void visit(readNode n) {
      // Compute address associated with target variable
      computeAdr(n.targetVar);
      // Call library routine to do the read
      if (n.targetVar.varName.idinfo.type == Integer)
            genCall("CSXLib/readInt()I");
      else // targetVar.varName.idinfo.type == Character
            genCall("CSXLib/readChar()C");
      storeName(n.targetVar);
      this.visit(n.moreReads);
}
```

For write statements, we'll first evaluate an operand, placing an integer, character, boolean or string value, or character array reference, on the stack. Then we'll call the appropriate library routine, based on the type of the operand. We'll update the code generation routine for `nameNodes` to include array values.

```
String arrayTypeCode(Types type){
         // Return array type code
         switch(type){
               case Integer: return "[I";
               case Character: return "[C";
               case Boolean: return "[Z"; }
}

void loadGlobalReference(String name, String typeCode){
    // Generate a load of a reference to the stack from
    // a static field:
    //   getstatic CLASS/name typeCode
}

void loadLocalReference(int index){
    // Generate a load of a reference to the stack from
    // a local variable:
    //    aload index
}

void visit(nameNode n) {
    if (n.subscriptVal.isNull()) {
          // Simple (unsubscripted) identifier
          if (n.varName.idinfo.kind == Kinds.Var ||
              n.varName.idinfo.kind == Kinds.Value ||
              n.varName.idinfo.kind == Kinds.ScalarParm) {
                // id is a scalar variable, parameter or const
                if (n.varName.idinfo.adr == Global){
                      // id is a global
                      String label = n.varName.idinfo.label;
                      loadGlobalInt(label);
                } else { // (n.varName.idinfo.adr == Local)
                      n.varIndex = n.varName.idinfo.varIndex;
                      loadLocalInt(n.varIndex);
          } } else  { // varName is an array var or array parm
                if (n.varName.idinfo.adr == Global){
                      n.label = n.varName.idinfo.label;
                      loadGlobalReference(n.label,
                            arrayTypeCode(n.varName.idinfo.type));
                } else { // (n.varName.idinfo.adr == local)
                      n.varIndex = n.varName.idinfo.varIndex;
                      loadLocalReference(n.varIndex);
          }   }
          n.adr = stack;
    } else {} // Handle subscripted variables later
 }

 void visit(strLitNode n) {
    // generate:
    //  ldc n.strval
 }
 void visit(printNode n){
```

```
    // Compute value to be printed onto stack
    this.visit(n.outputValue);
    if (n.outputValue.kind == Kinds.Array ||
        n.outputValue.kind == Kinds.ArrayParm)
         genCall("CSXLib/printCharArray([C)V");
    else if (n.outputValue.kind == Kinds.String)
         genCall("CSXLib/printString(Ljava/lang/String;)V");
    else switch(n.outputValue.type){
         case Integer:
              genCall("CSXLib/printInt(I)V");
              break;
         case Boolean:
              genCall("CSXLib/printBool(Z)V");
              break;
         case Character:
              genCall("CSXLib/printChar(C)V");
              break;
         }
    this.visit(n.morePrints);
}
```

While Loops

The JVM code we will generate will be of the form:

```
L1: {Evaluate control expr onto stack}
    ifeq L2  ; branch to L2 if top of stack == 0 (false)
    {Code for loop body}
    goto L1
L2:
```

We will assume that the `SymbolInfo` node contains two `String` valued fields, `topLabel` and `bottomLabel`. These will be used for identifiers that label while loops. The code needed to translate a `whileLoopNode` is:

```
static int labelCnt = 1;

String genLab(){
   return "L"+labelCnt++;
}

void defineLab(String label){
   // Generate:
   // label:
}

void branchZ(String label){
   // Generate branch to label if stack top contains 0:
   //    ifeq label
}
```

```
void branch(String label) {
   // Generate unconditional branch to label:
   // goto label
}

void visit(whileNode n) {
   String top = genLab();
   String bottom = genLab();
   if (! n.label.isNull()){
         ((identNode) n.label).idinfo.topLabel = top;
         ((identNode) n.label).idinfo.bottomLabel = bottom;}
   defineLab(top);
   this.visit(n.condition);
   branchZ(bottom);
   this.visit(n.loopBody);
   branch(top);
   defineLab(bottom);
}
```

Break and Continue Statements

Break and continue statements will use the bottom-of-loop and top-of-loop labels stored in the `SymbolInfo` nodes associated with labels.

```
void visit(breakNode n) {
   branch(n.label.idinfo.bottomLabel);
}

void visit(continueNode n) {
   branch(n.label.idinfo.topLabel);
}
```

Conditional Statements

The JVM code we will generate will be of the form:

```
   {Evaluate control expr onto stack top}
   ifeq L1
   {Code for then part}
   goto L2
L1:
   {Code for else part}
L2:
```

The code to translate an `ifThenNode` is:

```
void visit(ifThenNode n) {
   String endLab;   // label that will mark end of if stmt
   String elseLab;  // label that will mark start of else part
   // translate boolean condition, pushing it onto the stack
   this.visit(n.condition);
   elseLab = genLab();
   // generate conditional branch around then stmt
   branchZ(elseLab);
   // translate then part
   this.visit(n.thenPart);
   // branch around else part
   endLab = genLab();
   branch(endLab);
   // translate else part
   defineLab(elseLab);
   this.visit(n.elsePart);
   // generate label marking end of if stmt
   defineLab(endLab);
}
```

Type Casts and Relational Operators

The JVM does not include operations that directly implement relational operators (`==`, `!=`, etc.). Rather they must be synthesized using conditional branches.

```
String relationCode(int tokenCode){
//Determine relation code of an op based on its tokenCode:
   switch(tokenCode){
         case sym.EQ:    return "eq";
         case sym.NOTEQ: return "ne";
         case sym.LT:    return "lt";
         case sym.LEQ:   return "le";
         case sym.GT:    return "gt";
         case sym.GEQ:   return "ge";
         default:        return "";
} }

void branchRelationalCompare(int tokenCode, String label){
// Generate a conditional branch to label based on tokenCode:
// Generate:
// "if_icmp"+relationCode(tokenCode)  label
}
```

```
void genRelationalOp(int operatorCode){
// Generate code to evaluate a relational operator
   String trueLab = genLab();
   String skip = genLab();
   branchRelationalCompare(operatorCode, trueLab);
   loadI(0); // Push false
   branch(skip);
   defineLab(trueLab);
   loadI(1); // Push true
   defineLab(skip);
}
```

We update our `visit` method for `binaryOpNodes`:

```
void visit(binaryOpNode n) {
   // First translate the left and right operands
   this.visit(n.leftOperand);
   this.visit(n.rightOperand);
   // Now the values of the operands are on the stack
   // Is this a relational operator?
   if (relationCode(n.operatorCode) == ""){
        gen(selectOpCode(n.operatorCode));
   } else { // relational operator
        genRelationalOp(n.operatorCode);
   }
   n.adr = Stack;
}
```

For a `castNode`, we need to generate code for only two cases. If an `int` or `char` value is cast into a `bool`, we must generate code to test if the value is not equal to zero. If an `int` is cast into a `char`, we must extract the rightmost 7 bits of the integer value. In all other cases, the value of the operand may be used without modification.

```
void visit(castNode n) {
   // First translate the operand
   this.visit(n.operand);
   // Is the operand an int or char and the resultType a bool?
   if ( ((n.operand.type == Integer) ||
         (n.operand.type == Character)) &&
         (n.resultType instanceof boolTypeNode)){
             loadI(0);
             genRelationalOp(sym.NOTEQ);
   } else if ((n.operand.type == Integer) &&
             (n.resultType instanceof charTypeNode)){
                  loadI(127); // Equal to 1111111B
                  gen("iand");
   }
}
```