CS 536 — Spring 2008

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 call `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
}
```

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

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 `nameNode`s to include array values.

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

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

cg(){ // for strLitNode
   // generate:
   //  ldc strval
}
```

```
cg(){ // for writeNode
   outputValue.cg();
   if (outputValue.kind.val == Kinds.Array ||
       outputValue.kind.val == Kinds.ArrayParm)
      genCall("CSXLib/printCharArray([C)V");
   else switch(outputValue.type.val){
      case Types.Integer:
         genCall("CSXLib/printInt(I)V");
         break;
      case Types.Boolean:
         genCall("CSXLib/printBool(Z)V");
         break;
      case Types.Character:
         genCall("CSXLib/printChar(C)V");
         break;
      case Types.String:
         genCall("CSXLib/printString(Ljava/lang/String;)V");
         break;
    }
    moreWrites.cg();
}
```

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 nconditional branch to label:
   // goto label
}

cg(){ // for whileLoopNode
   String top = genLab();
   String bottom = genLab();
   if (! label.isNull()){
      label.idinfo.topLabel = top;
      label.idinfo.bottomLabel = bottom;}
   defineLab(top);
   condition.cg();
   branchZ(bottom);
   loopBody.cg();
   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. The JVM code we will generate is:

```
cg(){ // for breakNode
   branch(label.idinfo.bottomLabel);
}

cg(){ // for continueNode
   branch(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:

```
    cg(){ // for ifThenNode
       condition.cg();
       String elseLab = genLab();
       branchZ(elseLab);
       thenPart.cg();
       String endLab = genLab();
       branch(endLab);
       defineLab(elseLab);
       elsePart.cg();
       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 operator 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 cg() for binaryOpNodes:

```
cg(){// for binaryOpNode
   // First translate the left and right operands
   leftOperand.cg();
   rightOperand.cg();
   adr = stack;
   if(relationCode(operatorCode) == "")
      // Ordinary (non relational) operator
      binOp(selectOpCode(operatorCode));
   else // relational operator
      genRelationalOp(operatorCode);
}
```

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.

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

Indexing and Assigning Arrays

The JVM includes special instructions for loading from and storing into arrays. Integer arrays use iaload and iastore. Boolean arrays use baload and bastore. Character arrays use caload and castore.

To assign arrays we'll use the CSXLib methods

```
int [] cloneIntArray(int[]),
boolean[] cloneBoolArray(boolean[]),
char[] cloneCharArray(char[]),
char[] convertString(String),
```

`int []` `checkIntArrayLength(int[], int[])`,
`boolean[]` `checkBoolArrayLength(boolean[], boolean[])`,
`char[]` `checkCharArrayLength(char[], char[])`.

These routines make a copy (clone) of the source array and check the length of the source and target arrays (in case an array parameter is involved). If the array lengths are not compatible, an `ArraySizeException` is raised.

We'll extend `computeAdr`, `storeName` and `cg` for `nameNode`s and `asgNode`s to include array indexing and assignment.

```
cg(){ // for nameNode (final revision)
   adr = stack;
   if (subscriptVal.isNull()) {
      // Simple (unsubscripted) identifier
      if (varName.idinfo.kind.val == Kinds.Var ||
          varName.idinfo.kind.val == Kinds.Value ||
          varName.idinfo.kind.val == Kinds.ValueParm) {
        // id is a scalar variable, parameter or const
         if (varName.idinfo.adr == global){// id is a global
            label = varName.idinfo.label;
            loadGlobalInt(label);
         } else { // (varName.idinfo.adr == local)
            varIndex = varName.idinfo.varIndex;
            loadLocalInt(varIndex);
      }  } else { // varName is an array var or array parm
            if (varName.idinfo.adr == global){
               label = varName.idinfo.label;
               loadGlobalReference(label,
                    arrayTypeCode(varName.idinfo.type));
            } else { // (varName.idinfo.adr == local)
               varIndex = varName.idinfo.varIndex;
               loadLocalReference(varIndex);
      }     }
   } else { // This is a subscripted variable
            // Push array reference first
            if (varName.idinfo.adr == global){
               label = varName.idinfo.label;
               loadGlobalReference(label,
                    arrayTypeCode(varName.idinfo.type));
            } else { // (varName.idinfo.adr == local)
               varIndex = varName.idinfo.varIndex;
               loadLocalReference(varIndex);
            } // Next compute subscript expression
            subscriptVal.cg();

            // Now load the array element onto the stack
            switch(type.val){
               case Types.Integer:
                 // generate: iaload
                 break;
               case Types.Boolean:
```

```
                // generate: baload
                break;
              case Types.Character:
                // generate: caload
                break;
} }         }


    void computeAdr(nameNode name) { // revised
    // Compute address associated w/ name node
    // don't load the value addressed onto the stack
       if (name.subscriptVal.isNull()) {
          // Simple (unsubscripted) identifier
          if (name.varName.idinfo.kind.val == Kinds.Var ||
              name.varName.idinfo.kind.val == Kinds.ValueParm) {
          // id is a scalar variable
             if (name.varName.idinfo.adr == global) {
                name.adr = global;
                name.label = name.varName.idinfo.label;
             } else { // varName.idinfo.adr == local
                name.adr = local;
                name.varIndex = name.varName.idinfo.varIndex;
          }} else { // Must be an array
                // Push ref to target array to check length
                if (name.varName.idinfo.adr == global){
                   label = name.varName.idinfo.label;
                   loadGlobalReference(label,
                        arrayTypeCode(name.varName.idinfo.type));
                } else { // (name.varName.idinfo.adr == local)
                   varIndex = name.varName.idinfo.varIndex;
                   loadLocalReference(varIndex);
             }  }
       } else { // This is subscripted variable
                // Push array reference first
                if (name.varName.idinfo.adr == global){
                   label = name.varName.idinfo.label;
                   loadGlobalReference(label,
                        arrayTypeCode(name.varName.idinfo.type));
                } else { // (name.varName.idinfo.adr == local)
                   varIndex = name.varName.idinfo.varIndex;
                   loadLocalReference(varIndex);
                } // Next compute subscript expression
                name.subscriptVal.cg();
    }  }
```

```
void storeName(nameNode name) { // revised
   if (name.subscriptVal.isNull()) {
      // Simple (unsubscripted) identifier
      if (name.varName.idinfo.kind.val == Kinds.Var ||
          name.varName.idinfo.kind.val == Kinds.ValueParm) {
         if (name.adr == global)
             storeGlobalInt(name.label);
         else // (name.adr == local)
             storeLocalInt(name.varIndex);
      } else { // Must be an array
         // Check the lengths of the source and target arrays
         switch(name.type.val){
            case Types.Integer:
              genCall("CSXLib/checkIntArrayLength([I[I)[I");
              break;
            case Types.Boolean:
              genCall("CSXLib/checkBoolArrayLength([Z[Z)[Z");
              break;
            case Types.Character:
              genCall("CSXLib/checkCharArrayLength([C[C)[C");
              break;
         }  // Now store source array away in target var
         if (name.varName.idinfo.adr == global){
            label = name.varName.idinfo.label;
            storeGlobalReference(label,
                 arrayTypeCode(name.varName.idinfo.type));
         } else { // (name.varName.idinfo.adr == local)
            varIndex = name.varName.idinfo.varIndex;
            storeLocalReference(varIndex);
      }  }
   } else  // This is a subscripted variable
           // A reference to the target array, the
           //  subscript expression and the source expression
           //  have already been pushed.
           // Now store the source value into the array
           switch(type.val){
               case Types.Integer:
                 // generate: iastore
                 break;
               case Types.Boolean:
                 // generate: bastore
                 break;
               case Types.Character:
                 // generate: castore
                 break;
            }
}
```

```
cg(){ // for asgNode (extended)
   // Compute address associated with LHS
   computeAdr(target);
   // Translate RHS (an expression)
   source.cg();
   // Check to see if source needs to be cloned or converted
   if (source.kind.val == Kinds.Array ||
       source.kind.val == Kinds.RefArray)
          switch(source.type.val){
             case Types.Integer:
              genCall("CSXLib/cloneIntArray([I)[I");
              break;
            case Types.Boolean:
              genCall("CSXLib/cloneBoolArray([Z)[Z");
              break;
            case Types.Character:
              genCall("CSXLib/cloneCharArray([C)[C");
              break;
          }
   else if (source.type.val == Types.String)
            genCall(
              "CSXLib/convertString(Ljava/lang/String;)[C");

   // Finally, store source into the target (a name)
   storeName(target);
}
```