CS 536 — Spring 2015

CSX Code Generation Routines

Part IV

CSX Class Body

We first generate the Jasmin class header declarations. A shared field `CLASS` is set to the name of the CSX class. Next, we generate field declarations. We then generate a method defintion for `main(String [])`, which is the standard starting point for JVM executions. This method contains non-trivial field initializations and a call to `main()`, the standard CSX starting point. Finially, methods are translated.

```
void visit(classNode n) {
   currentMethod = null; // We're not in any method body (yet)
   CLASS = n.className.idname;
   // generate:
   //  .class public CLASS
   //  .super java/lang/Object

   // Generate field declarations for the class
   this.visit(n.members.fields);

   // generate:
   //  .method public static main([Ljava/lang/String;)V

   // Generate non-trivial field declarations
   this.visit(n.members.fields);

   // generate:
   //  invokestatic CLASS/main()V
   //  return
   // .limit stack 2
   // .end method

   // Finally translate methods
   this.visit(n.members.methods);
}
```

Methods

We'll assume that if we're currently translating an AST node that's within a method declaration (that is, "under" a `methodDeclNode`) then a shared field `currentMethod` points to that `methodDeclNode`. If we're not within a method declaration then `currentMethod` is `null`.

Calls are straightforward to translate. We first translate the parameter list, pushing each actual parameter onto the stack. We then "call" the appropriate method. Because Jasmin requires that a method name in a call must contain type codes for the parameters and return value, we will build the appropriate code using the overloaded utility routine `buildTypeCode`. If an integer function `f` is called with a single integer parameter, it has a type code code of "`f(I)I`". Since we look at the types of the parameters actually used in the call, overloaded methods are correctly handled.

```
String typeCode(typeNode type){
      // Return type code
      if (type instanceof intTypeNode)
            return "I";
      else if (type instanceof charTypeNode)
            return "C";
      else if (type instanceof boolTypeNode)
            return "Z";
      else // (type instanceof voidTypeNode)
            return "V";
}

String typeCode(Types type){
      // Return type code
      switch (type){
            case Integer:
                  return "I";
            case Character:
                  return "C";
            case Boolean:
                  return "Z";
            case Void:
                  return "V";
      }
}

String buildTypeCode(argDeclNode n) {
      if (n instanceof valArgDeclNode)
            return typeCode(((valArgDeclNode) n).argType);
      else // must be an arrayArgDeclNode
            return
             arrayTypeCode(((arrayArgDeclNode) n).elementType);
}
```

```
String buildTypeCode(argDeclsNode n){
      if (n.moreDecls.isNull())
            return buildTypeCode(n.thisDecl);
      else
            return buildTypeCode(n.thisDecl)+
                   buildTypeCode((argDeclsNode) n.moreDecls);
}

String buildTypeCode(exprNode n){
      if (isArray(n.kind))
            return arrayTypeCode(n.type);
      else return typeCode(n.type);
}

String buildTypeCode(argsNode n){
      if (n.moreArgs.isNull())
            return buildTypeCode(n.argVal);
      else return buildTypeCode(n.argVal) +
                  buildTypeCode((argsNode) n.moreArgs);

}

String buildTypeCode(String methodName,
                     argsNodeOption args, String returnCode){
      String newTypeCode = methodName;
      if (args.isNull())
            newTypeCode = newTypeCode + "()";
      else newTypeCode = newTypeCode + "(" +
                           buildTypeCode((argsNode) args) + ")";
      return newTypeCode + returnCode;
}


void visit(argsNode n) {
      // Evaluate arguments and load them onto stack
      this.visit(n.argVal);
      this.visit(n.moreArgs);
}

void visit(callNode n) {
      // Evaluate args and push them onto the stack
      this.visit(n.args);
      // Generate call to method, using its type code
      String typeCode =
          buildTypeCode(n.methodName.idname, n.args,
                        n.methodName.idinfo.methodReturnCode);
      genCall(CLASS+ "/" + typeCode);
}
```

The translation of `fctCallNodes` is essentially identical to that of `callNodes`.

Return statements within a function will evaluate the return value onto the stack and then do an `ireturn`. Return statements within a procedure generate a `return`.

```
void visit(returnNode n) {
      if (n.returnVal.isNull())
            // generate: return
      else { // Evaluate return value
            this.visit(n.returnVal);
            // generate: ireturn
      }
}
```

Within methods, each parameter and local variable will be assigned a "local variable index," starting at zero. A field `numberOfLocals`, within a method's `SymbolInfo` node, will track how many locals have been allocated an index.

```
void visit(argDeclsNode n) {
      // Label each method argument with its address info
      this.visit(n.thisDecl);
      this.visit(n.moreDecls);
}

void visit(valArgDeclNode n) {
      // Label method argument with its address info
      n.argName.idinfo.adr = local;
      n.argName.idinfo.varIndex =
            currentMethod.name.idinfo.numberOfLocals++;
}

void visit(arrayArgDeclNode n) {
      // Label method argument with its address info
      n.argName.idinfo.adr = local;
      n.argName.idinfo.varIndex =
            currentMethod.name.idinfo.numberOfLocals++;
}
void visit(methodDeclNode n) {
      currentMethod = n; // We're in a method now!
      n.name.idinfo.numberOfLocals = 0;
      String newTypeCode = n.name.idname;
      if (n.args.isNull())
            newTypeCode = newTypeCode + "()";
      else newTypeCode = newTypeCode + "(" +
               buildTypeCode((argDeclsNode) n.args) + ")";
      newTypeCode = newTypeCode + typeCode(n.returnType);
      n.name.idinfo.methodReturnCode = typeCode(n.returnType);
      // generate:
      // .method public static newTypeCode
      this.visit(n.args); // Assign local variable indices to args
      // Generate code for local decls and method body
      this.visit(n.decls);
      this.visit(n.stmts);
```

```
      // generate default return at end of method body
      if (n.returnType instanceof voidTypeNode)
            // generate: return
      else { // Push a default return value of 0
            loadI(0);
            // generate: ireturn
      }
      // Generate end of method data;
      // we'll guestimate stack depth needed at 25
      //  (almost certainly way too big!)
      // generate: .limit stack 25
      // generate: .limit locals n.name.idinfo.numberOfLocals
      // generate: .end method
}
```

We need to extend `visit` methods for `varDeclNode`, `constDeclNode` and `arrayDeclNode` to handle local variables.

```
void visit(varDeclNode n){
      if (currentMethod == null) // A global field decl
            if (n.varName.idinfo.adr == none)
                  // First pass; generate field declarations
                  declField(n);
            else { // 2nd pass; do field init (if needed)
                  if (! n.initValue.isNull())
                        if (! isNumericLit(n.initValue)) {
                              // Compute init value & store in field
                              this.visit(n.initValue);
                              storeId(n.varName);
            }           }
      else {// Process local variable declaration
            //   Give this var an index equal to numberOfLocals
            //     and remember index in symbol table entry
             n.varName.idinfo.varIndex =
                  currentMethod.name.idinfo.numberOfLocals;
             n.varName.idinfo.adr = local;
             //   Increment numberOfLocals used in this method
             currentMethod.name.idinfo.numberOfLocals++;
             // Do initialization (if necessary)
             if (! n.initValue.isNull()){
                  this.visit(n.initValue);
                  storeId(n.varName);
}     }     }
```

```
void visit(constDeclNode n) {
     if (currentMethod == null) // A global const decl
          if (n.constName.idinfo.adr == none)
               // First pass; generate field declarations
               declField(n);
          else {// 2nd pass; do field initialization (if needed)
               if (! isNumericLit(n.constValue)) {
                    // Compute init value & store in field
                    this.visit(n.constValue);
                    storeId(n.constName);
          }         }
     else {// Process local const declaration
          // Give this variable an index equal to numberOfLocals
          //     and remember index in symbol table entry
          n.constName.idinfo.varIndex =
               currentMethod.name.idinfo.numberOfLocals;
          n.constName.idinfo.adr = local;
          //   Increment numberOfLocals used in this method
          currentMethod.name.idinfo.numberOfLocals++;
          // compute and store const value
          this.visit(n.constValue);
          storeId(n.constName);
     }
}

void visit(arrayDeclNode n) {
     // Create a new array and store resulting reference
     if (currentMethod == null) { // A global array decl
          if (n.arrayName.idinfo.adr == none)  {
               // First pass; generate field declarations
               declField(n);
               return;
          }
     } else {
          // Process local array declaration
          // Give this variable an index equal to numberOfLocals
          //     and remember index in symbol table entry
          n.arrayName.idinfo.varIndex =
               currentMethod.name.idinfo.numberOfLocals;
           n.arrayName.idinfo.adr = local;
           //   Increment numberOfLocals used in this method
           currentMethod.name.idinfo.numberOfLocals++;
     }

     // Now create the array & store a reference to it
          loadI(n.arraySize.intval); // Push size of array
          allocateArray(n.elementType);
          if (n.arrayName.idinfo.adr == global)
               storeGlobalReference(n.arrayName.idinfo.label,
                    arrayTypeCode(n.elementType));
          else storeLocalReference(n.arrayName.idinfo.varIndex);
}
```

Finally, we handle blocks. Local declarations are allocated within the frame of the enclosing method.

```
void visit(blockNode n) {
// Generate code for block decls and body
      this.visit(n.decls);
      this.visit(n.stmts);
}
```