

CS 536 — Spring 2008

CSX Code Generation Routines

Part III

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 definition 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. Finally, methods are translated.

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

    // generate field declarations for the class
    members.fields.declField();

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

    // generate non-trivial field initializations
    members.fields.cg();

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

    // Finally translate methods
    members.methods.cg();
}
```

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 require that each method, when translated, creates a field `methodTypeCode`, in its `SymbolInfo` node that contains this type code. Thus an integer function `f` that takes a single integer parameter, would have a code of "`f(I)I`".

```
cg(){ for argsNode
// Evaluate arguments and load them onto stack
    argVal.cg();
    moreArgs.cg();
}

cg(){ // for callNode
// Evaluate args and push them onto the stack
    args.cg();

// Generate call to method, using its type code
    genCall(CLASS+"/"+methodName.idinfo.methodTypeCode);
}
```

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`.

```
cg(){ // for returnNode
    if (returnVal.isNull())
        // generate: return
    else { // Evaluate return value
        returnVal.cg();
        // generate: ireturn
    }
}
```

Jasmin requires that a method name in a call must contain type codes for the parameters and return value, so we'll add a method `buildTypeCode()` to `argDeclsNode`, `argDeclNode` (as an abstract method), `valArgDeclNode` and `arrayArgDeclNode`. This will allow us to easily compute and store the necessary type code within a method.

```
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 buildTypeCode() { // for argDeclsNode
    return thisDecl.buildTypeCode()+moreDecls.buildTypeCode();
}
```

```

String buildTypeCode() { // for valArgDeclNode
    return typeCode(argType);
}

String buildTypeCode() { // for arrayArgDeclNode
    return arrayTypeCode(argType);
}

```

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.

```

cg(){ // for methodDeclNode
    currentMethod = this; // We're in a method now!
    methodName.idinfo.numberOfLocals = 0;
    String newTypeCode = name.idname;
    if (args.isNull())
        newTypeCode = newTypeCode + "()";
    else newTypeCode =
        newTypeCode + "(" + args.buildTypeCode() + ")";
    newTypeCode =
        newTypeCode + typeCode(returnType);
    methodName.idinfo.methodTypeCode = newTypeCode;

    // generate:
    //   .method public static newTypeCode

    args.cg(); // Assign local variable indices to args
    // Generate code for local decls and method body
    decls.cg();
    stmts.cg();

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

```

```

cg(){ // for argDeclsNode
// Label each method argument with its address info
  thisDecl.cg();
  moreDecls.cg();
}

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

cg(){ // for arrayArgDeclNode
// Label method argument with its address info
  argName.idinfo.adr = local;
  argName.idinfo.varIndex =
    currentMethod.name.idinfo.numberOfLocals++;
}

```

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

```

cg(){ // for varDeclNode
  if (currentMethod == null) { // A global var declaration
    // Same as defined previously
    if(!(initValue.isNull()) &&
        !(initValue instanceof intLitNode) {
      // compute and assign initial value
      initValue.cg();
      storeId(varName);
    } }
  else { // A local var declaration
    varName.idinfo.adr = local;
    varName.idinfo.varIndex =
      currentMethod.name.idinfo.numberOfLocals++;
    if(!initValue.isNull()) { // assign init value
      initValue.cg();
      storeId(varName);
    } } }
}

```

```

cg(){ // for constDeclNode
    if (currentMethod == null) { // A global const declaration
        // Same as defined previously
        if(!(constValue instanceof intLitNode) {
            // compute and assign initial value
            constValue.cg();
            storeId(constName);
        } }
    else { // A local const
        constName.idinfo.adr = local;
        constName.idinfo.varIndex =
            currentMethod.name.idinfo.numberOfLocals++;
        constValue.cg();
        storeId(constName);
    } }
}

```

```

cg(){ // for arrayDeclNode
    // Create a new array and store resulting reference
    if (currentMethod != null) { // A local array declaration
        arrayName.idinfo.adr = local;
        arrayName.idinfo.varIndex =
            currentMethod.name.idinfo.numberOfLocals++;
    }
    // Now create the array & store a reference to it
    loadI(arraySize.intval); // Push number of array elements
    allocateArray(elementType);
    if (arrayName.idinfo.adr == global)
        storeGlobalReference(arrayName.idinfo.label,
            arrayTypeCode(elementType));
    else storeLocalReference(arrayName.idinfo.varIndex);
}

```

Finally, we handle blocks.

```

cg(){ // for blockNode
    // Generate code for block decls and body
    decls.cg();
    stmts.cg();
}

```