CS 536 — Spring 2015

CSX Code Generation Routines

Part I

All code generation methods are placed in class `CodeGenerating`. Each AST node has an associated `visit` method that translates the node into JVM assember code. A variety of useful auxiliary methods are also included.

To begin you create an instance of class `CodeGenerating` by calling the constructor

```
new CodeGenerating(PrintStream asmFile)
```

The file parameter is the file into which JVM instructions are to be written. You then call the boolean-valued method `startCodeGen(root)` where root is the `root` of the AST built by the parser. This method will begin traversal of the AST, generating JVM code into `asmFile`.

If any errors are detected during code generation, `startCodeGen` will return `false`; the contents of `asmFile` need not be valid. If no errors are detected by the code generator, `true` is returned and the contents of `asmFile` will be a valid JVM assembly program that can be assembled using `jasmin`.

We'll focus on the content of `visit` methods needed for various AST nodes, along with miscellaneous useful subroutines. We'll group the code generation methods based on the kind of constructs being considered (expressions, declarations, conditional statements, looping statements, etc.). We'll start with simple constructs and work our way toward the more complex ones.

Addressing Values

The main purpose of code generation is to compute the values specified by the source program. Hence tracking exactly *where* values are is very important. Data values may be accessed (addressed) in many ways. They may be *global* and accessed in memory via a field label (that the assembler translates into a static field address). *Local* data is accessed in a frame, using a local variable index. Elements of an array are addressed using an array reference and an index. Constant values may be literals, in which case their values may not be in memory at all (unless we put them there). Values may be on the stack as well as in memory. In fact some values (like expression values) may reside *only* on the stack.

To keep all these possibilities straight, we'll introduce the following enumeration values:

```
global, local, stack, literal, none
```

We'll add a field

```
AdrMode adr; // One of global, local, stack, literal, none
```

to expression nodes and to the `SymbolInfo` class associated with all identifiers. If an AST node or an identifier denotes a value, `adr` will tell us how it may be accessed.

Based on the value of `adr`, additional fields in the AST node (and `SymbolInfo` class) will provide additional information on how the value is to be accessed:

adr value	**Additional fields**	**Comments**
`global`	`String label;`	// Label used for global field names
`local`	`int varIndex;`	// Index of local variable
`literal`	`int intval;` `String strVal;`	// Value of int, char or bool literals // Value of string literals
`stack`		// No fields used. Value is on stack.
`none`		// No fields used. AST node has no value // (type == void)

Expressions

Expressions will be computed onto the JVM operand stack. We'll use the following subroutines in translating expressions (and other constructs). Assume `CLASS` contains the name of the CSX class being compiled.

```
void loadI(int val){
// Generate a load of an int literal:
//     ldc val
}

void loadGlobalInt(String name){
// Generate a load of an int static field onto the stack:
//     getstatic CLASS/name I
}

void loadLocalInt(int index){
// Generate a load of an int local variable onto the stack:
//     iload index
}


void binOp(String op){
// Generate a binary operation;
// all operands are on the stack:
//     op
}
```

```
void storeGlobalInt(String name){
// Generate a store into an int static field from the stack:
//    putstatic CLASS/name

void storeLocalInt(int index){
// Generate a store to an int local variable from the stack:
//    istore index
}
```

Code generation definitions for simple AST nodes that appear in expression trees appear below.

```
visit(intLitNode n) {
  loadI(n.intval);
  n.adr = literal;
}

visit(charLitNode n) {
  loadI(n.charval);
  n.adr = literal;
  n.intval = n.charval;
}

visit(trueNode n) {
  loadI(1);
  n.adr = literal;
  n.intval = 1;
}

visit(falseNode n) {
  loadI(0);
  n.adr = literal;
  n.intval = 0;
}

visit(nameNode n) {
  if (n.subscriptVal.isNull()) {
     // Simple (unsubscripted) identifier
     if (n.varName.idinfo.kind == Kinds.Var ||
          n.varName.idinfo.kind == Kinds.Value) {
              // id is a scalar variable or const
              if (n.varName.idinfo.adr == global){
                   // id is a global
                   label = n.varName.idinfo.label;
                   loadGlobalInt(label);
              } else { // (n.varName.idinfo.adr == Local)
                   n.varIndex = n.varName.idinfo.varIndex;
                   loadLocalInt(n.varIndex);
     } } else   // Handle arrays later
     n.adr = stack;
  } else {} // Handle subscripted variables later
}
```

```
String selectOpCode(int tokenCode){
  switch (tokenCode) {
        case sym.PLUS: return "iadd";
        case sym.MINUS: return "isub";
  // Remaining CSX operators are handled here
  }
}
visit (binaryOpNode n) {
  // First translate the left and right operands
        this.visit(n.leftOperand);
        this.visit(n.rightOperand);
        binOp(selectOpCode(n.operatorCode));
        n.adr = stack;
  }
```

Assignment Statements

We'll first define useful subroutines, and then show how to translate an `asgNode`.

```
void computeAdr(nameNode name) {
  // 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 == Kinds.Var) {
       // 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// Handle arrays later
  } else  // Handle subscripted variables later
}

void storeId(identNode id) {
   if (id.idinfo.kind == Kinds.Var ||
       id.idinfo.kind == Kinds.Value ) {
   // id is a scalar variable
      if (id.idinfo.adr == global) // ident is a global
         storeGlobalInt(id.idinfo.label);
      else // (id.idinfo.adr == local)
         storeLocalInt(id.idinfo.varIndex);
   } else  // Handle arrays later
}
```

```
void storeName(nameNode name) {
   if (name.subscriptVal.isNull()) {
      // Simple (unsubscripted) identifier
      if (name.varName.idinfo.kind == Kinds.Var) {
         if (name.adr == global)
             storeGlobalInt(name.label);
         else // (name.adr == local)
             storeLocalInt(name.varIndex);
       } else  // Handle arrays later

   } else  // Handle subscripted variables later
}


void visit(asgNode n){
   // Compute address associated with LHS
   computeAdr(n.target);
   // Translate RHS (an expression) onto stack
   this.visit(n.source);
   // Then store it into LHS
   storeName(n.target);
  }
```

Global Field Declarations

We'll handle the translation of field names by calling the code generator *twice*. First, Jasmin field declarations are produced. The code generator uses an auxiliary method `declField`(n). After field declarations are produced, a second call to the code generator finishes global field translation by doing necessary non-trivial initializations.

Jasmin requires that labels not clash with JVM operation codes, so we'll append a '`$`' to each field name to create the label used to access that field. For simplicity and uniformity, character and boolean fields will be declared as integers rather than bytes or bits.

```
// Is this expression a trivial numeric literal?
boolean isNumericLit(exprNodeOption e){
       return  (e instanceof intLitNode) ||
                (e instanceof charLitNode) ||
                (e instanceof trueNode) ||
                (e instanceof falseNode);     }

int getLitValue(exprNode e){
       if  (e instanceof intLitNode)
             return e.intval;
       else if (e instanceof charLitNode)
             return ((charLitNode) e).charval;
       else if (e instanceof trueNode)
             return 1;
       else if (e instanceof falseNode)
             return 0; }
```

```
void declGlobalInt(String name, exprNodeOption initValue){
  if (isNumericLit(initValue))
        numValue = getLitValue(initValue);
        // Generate a field declaration with initial value:
        // .field public static name I = numValue
  else
       // Gen a field declaration without an initial value:
       // .field public static name I
}

String arrayTypeCode(typeNode type){
  // Return array type code
  if (type instanceof intTypeNode)
      return "[I";
  else if (type instanceof charTypeNode)
      return "[C";
  else // (type instanceof boolTypeNode)
      return "[Z";

}


void declGlobalArray(String name, typeNode type){
   // Generate a field declaration for an array:
   // .field public static name arrayTypeCode(type)
}


void allocateArray(typeNode type){
   if (type instanceof intTypeNode)
      // Generate a newarray instruction for an integer array:
      // newarray int
   else if (type instanceof charTypeNode)
      // Gen a newarray instruction for a character array:
      // newarray char
   else // (type instanceof boolTypeNode)
        // Gen a newarray instruction for a boolean array:
        // newarray boolean

}
void storeGlobalReference(String name, String typeCode){
    // Generate a store of a reference from the stack into
    // a static field:
    //    putstatic CLASS/name typeCode
}

void storeLocalReference(int index){
    // Generate a store of a reference from the stack into
    // a local variable:
    //    astore index
}
```

```
declField(varDeclNode n){
   String varLabel = n.varName.idname +"$";
   declGlobalInt(varLabel,n.initValue);
   n.varName.idinfo.label = varLabel;
   n.varName.idinfo.adr = global;
}

declField(constDeclNode n){
   String constLabel = n.constName.idname +"$";
   declGlobalInt(constLabel,n.constValue);
   n.constName.idinfo.label = constLabel;
   n.constName.idinfo.adr = global;
}

declField(arrayDeclNode n){
   String arrayLabel = n.arrayName.idname +"$";
   declGlobalArray(arrayLabel,n.elementType);
   n.arrayName.idinfo.label = arrayLabel;
   n.arrayName.idinfo.adr = global;
}

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 initialization (if needed)
         if (! n.initValue.isNull())
            if (! isNumericLit(n.initValue)) {
               // Compute init val onto stack; store in field
               this.visit(n.initValue);
               storeId(n.varName);
            }
      }
   else {// Handle local variable declarations later }
}


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 const val onto stack and store in field
            this.visit(n.constValue);
            storeId(n.constName);
         }
      }
   else {// Handle local const declarations later}
}
```

```
void visit(arrayDeclNode n) {
    if (currentMethod == null) {// A global array decl
      if (n.arrayName.idinfo.adr == none)  {
         // First pass; generate field declarations
         declField(n);
         return;}}
    else {// Handle local array declaration later}

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