CS 536 — Spring 1998

CSX Code Generation Routines

Part I

Code is generated by calling the member function

```
boolean codegen(PrintStream asmfile);
```

in the root of the AST (the root is always a `classNode`). This function saves the value of `asmfile` in the static field `afile` (so the file doesn't have to constantly be passed as a parameter to various code generation subroutines).

The member function

```
void cg();
```

does the actual code generation, on a per-AST node basis, writing the translation into the `afile`. We'll focus on the content of `cg()` definitions that will be needed for various AST nodes, along with miscellaneous useful subroutines. We'll group the `cg()` definitions and subroutines 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 constants:

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

We'll add a field

```
int 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 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 I
}

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

`cg` definitions for nodes that appear in expression trees appear below.

```
cg(){ // for intLitNode
   loadI(intval);
   adr = literal;
}

cg(){ // for charLitNode
   loadI(charval);
   adr = literal;
}

cg(){ // for trueNode
   loadI(1);
   adr = literal;
   intval = 1;
}

cg(){ // for falseNode
   loadI(0);
   adr = literal;
   intval = 0;
}

cg(){ // for nameNode
   if (subscriptVal.isNull()) {
      // Simple (unsubscripted) identifier
      if (varName.idinfo.kind.val == Kinds.Var ||
          varName.idinfo.kind.val == Kinds.Value) {
        // id is a scalar variable 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   // Handle arrays later
      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
     default: assert(FALSE); // Illegal operator
   }
}
```

```
cg(){// for binaryOpNode
   // First translate the left and right operands
   leftOperand.cg();
   rightOperand.cg();
   binOp(selectOpCode(operatorCode));
   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.val == 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.val == Kinds.Var ||
       id.idinfo.kind.val == 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.val == 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
}
```

```
cg(){ // for asgNode
   // Compute address associated with LHS
   computeAdr(target);
   // Translate RHS (an expression)
   source.cg();
   // Then store it into an ident
   storeName(target);
}
```

Global Field Declarations

We'll handle the translation of field names in two steps. First, Jasmin field declarations will be produced. This will be done by a member function `declField()` in `memberDeclsNode`, `declNode` (as an abstract member), `varDeclNode`, `constDeclNode` and `arrayDeclNode`.

After field declarations have been processed, calls to `cg()` will finish declarations for global fields by doing initializations that are necessary.

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.

```
void declGlobalInt(String name, exprNode initValue){
   if (initValue instanceof intLitNode)
      // Generate a field declaration with initial value:
      // .field public static name I = initvalue.intval
   else
      // Generate 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)
      // Generate a newarray instruction for a character array:
      // newarray char
   else // (type instanceof boolTypeNode)
      // Generate 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(){ // for varDeclNode

   String varLabel =
     CSXIdentifierToken.reg.toString(varName.idval)+"$";

   declGlobalInt(varLabel,initValue);

   varName.idinfo.label = varLabel;
   varName.idinfo.adr = global;
}

declField(){ // for constDeclNode

   String constLabel =
     CSXIdentifierToken.reg.toString(constName.idval)+"$";

   declGlobalInt(constLabel,constValue);

   constName.idinfo.label = constLabel;
   constName.idinfo.adr = global;
}
```

```
declField(){ // for arrayDeclNode

   String arrayLabel =
     CSXIdentifierToken.reg.toString(arrayName.idval)+"$";

   declGlobalArray(arrayLabel,elementType);

   arrayName.idinfo.label = arrayLabel;
   arrayName.idinfo.adr = global;
}


cg(){ // for varDeclNode
   if(!(initValue.isNull()) &&
      !(initValue instanceof intLitNode) {
     // compute and assign initial value
      initValue.cg();
      storeId(varName);
}  }

cg(){ // for constDeclNode
   if(!(constValue instanceof intLitNode) {
     // compute and assign initial value
      constValue.cg();
      storeId(constName);
}  }

cg(){ // for arrayDeclNode
   // Create a new array and store resulting reference

   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);
}
```