Code Generation – (2)
(Java Bytecode Generation)

Jeonggeun Kim
Kyungpook National University

This lecture's materials are based on Prof. Bernd Burgstaller's slides (CSI4104 Compiler Design, Yonsei University)

Lecture 4b: Java Bytecode Generation

1. Our MiniC to JVM Translation Model

2. Jasmin Assembly Code Generation
   - Expressions
   - Declarations
   - Short-circuit evaluation
   - Statements

Example 1: gcd.mc

```
int i = 2;
int j = 4;

int gcd (int a, int b) {
    while (b != 0) {
      if (a > b)
        a = a - b;
      else
        b = b - a;
    }
    return a;
}

int main() {
    putInt(gcd(i, j));
    return 0;  // optional in MiniC, C, C++
}
```

On the following slide, you see the Java code that does the same thing as the above MiniC program. The Jasmin code shows us what code we have to generate for the above program.

Example 1: gcd.mc versus gcd.java

```
int i = 2;
int j = 4;

int gcd (int a, int b) {
    while (b != 0) {
      if (a > b)
        a = a - b;
      else
        b = b - a;
    }
    return a;
}

int main() {
    putInt(gcd(i, j));
    return 0;
}
```

MiniC program

The red text is ``assumed'' by our MiniC compiler when we generate code.

- Make bytecode generated from MiniC AST look as it originated from a Java program.

```
import MiniC.lang.System;

public class gcd {
   static int i = 2;
   static int j = 4;

   public gcd(){} // constructor

   int gcd(int a, int b) {
      while (b != 0) {
        if (a > b)
          a = a - b;
        else
          b = b - a;
      }
      return a;
   }

   public static void main(
                String argv[]) {
      gcd mc$;
      mc$ = new gcd();
      System.putInt(mc$.gcd(i,j));
      return;
   }
}
```

Same MiniC program if it was written in Java

Code Generation Principle Idea:

```
int i = 2;
int j = 4;

int gcd (int a, int b) {
    while (b != 0) {
      if (a > b)
        a = a - b;
      else
        b = b - a;
    }
    return a;
}

int main() {
    putInt(gcd(i, j));
    return 0;
}
```

MiniC program

MiniC compiler

- MiniC compiler will generate the Jasmin bytecode **as if** it originated from the corresponding Java program compiled by the javac Java compiler.
- Following slides show how this is accomplished.

```
import MiniC.lang.System;

public class gcd {
   static int i = 2;
   static int j = 4;

   public gcd(){} // constructor

   int gcd(int a, int b) {
      while (b != 0) {
        if (a > b)
          a = a - b;
        else
          b = b - a;
      }
      return a;
   }

   public static void main(
                String argv[]) {
      gcd mc$;
      mc$ = new gcd();
      System.putInt(mc$.gcd(i,j));
      return;
   }
}
```

niC program if it was written in Java

javac

```
Jasmin bytecode
```

Example 1: gcd.mc (cont.)

- MiniC's **`int`** **`main() {...}`** is assumed to be

  `public static void main(String argv[ ]) {...}`

  - visit (FunDecl x): a **`return`** is always emitted just in case no return was present in the main function of a MiniC program.

  - visit (ReturnStmt): emit a **`return`** and not an **`ireturn`**, even if ``**`return expr`**" is present in the **`main()`** function of a MiniC program.

- All **MiniC functions** are assumed to be instance methods.

- All **global MiniC variables** are assumed to be static field variables.

- All **built-in functions** from the MiniC StdEnvironment are static.

Example 1: gcd.j

- MiniC **global variables** initialized by **class field initializer**:

```
int i = 2;
int j = 4;

int gcd (int a, int b) {...}

int main() {...}
```

```
.class public gcd
.super java/lang/Object

.field static i I
.field static j I

.method public <init>()V ;; gcd constructor
  .limit stack 1
  .limit locals 1
  .var 0 is this Lgcd
         ; from Label0 to Label1

  Label0:
.line 6
  0: aload_0
  1: invokespecial java/lang/Object/<init>()V

  Label1:
  4: return

.end method
```

```
; Class field initializer:
; called by the JVM after
; the class is loaded. Has
; to be present only if
; the class contains
; static fields with
; initial values.

.method static <clinit>()V
  .limit stack 1
  .limit locals 0

  .line 3
    0: iconst_2
    1: putstatic gcd.i I

  .line 4
    4: iconst_4
    5: putstatic gcd.j I
    8: return

.end method
```

Example 1: gcd.j (cont.)

```
.method  gcd(II)I
    .limit stack 2
    .limit locals 3
    .var 0 is this Lgcd; from Label2 to Label5
    .var 1 is arg0 I from Label2 to Label5
    .var 2 is arg1 I from Label2 to Label5
    Label2:
       0: iload_2
       1: ifeq Label0
       4: iload_1
       5: iload_2
       6: if_icmple Label1
       9: iload_1
      10: iload_2
      11: isub
      12: istore_1
      13: goto Label2
    Label1:
      16: iload_2
      17: iload_1
      18: isub
      19: istore_2
      20: goto Label2
    Label0:
      23: iload_1
    Label5:
      24: ireturn
.end method
```

- MiniC **functions** become **instance methods.**

Example 1: gcd.j (cont.)

```
.method public static main([Ljava/lang/String;)V
  Label0:
   new gcd
   dup
   invokespecial gcd/<init>()V
   astore_1
   ; CallStmt putInt: "this"-pointer is
   ; first ActualParam
   ; with instance method:
   aload_1 ; "this"-pointer
   ; ActualParam 2:
   getstatic gcd.i I
   ; ActualParam 3:
   getstatic gcd.j I
   invokevirtual gcd/gcd(II)I
   invokestatic lang/System/putInt(I)V
   ; ReturnStmt:
   return ; Was "return 0" from the user (see Slide 4).
  Label1: ; Compiler's default exit point from the procedure.
   return ; Dead-code elimination would remove 2nd return stmt.
   .limit locals 2
   .limit stack 150 ; We use a generous max stack size
.end method
```

```
public static void main(
               String argv[]) {
  gcd mc$;
  mc$ = new gcd();
  System.putInt(mc$.gcd(i,j));
  return;
}
```

A Visitor Code Generator

- We will program a Visitor for the AST class hierarchy.
- Syntax-driven: traverses the AST to emit code in pre-, in- or post-order, or any of their combinations.
- Classes:
  - Emitter.java: the visitor class for generating code
  - JVM.java: the class containing definitions regarding the JVM (Jasmin instructions from the JVM instruction set)
  - Frame.java: the class maintaining labels, local variable slots etc. for a function.

Code Templates

- **[[*X*]]**: a **code template** which defines the code generated for construct X
- Code template: a specification of [[*X*]] in terms of the codes for its syntactic components.
  - Code templates are **compositional**. Complex constructs (if statements, loops) defined in terms of its parts (conditions, statements etc.)
- Code templates will be provided to you
  - to be used for code generation in the Visitor code generator
  - Examples: see next slides.

Integer Literals

- Code Template: **[[*IntLiteral*]]:** emitICONST(IntLiteral.value)

```
private void emitICONST(int value) {
   if (value == -1)
      emit(JVM.ICONST_M1);
   else if (value >= 0 && value <= 5)
      emit(JVM.ICONST + "_" + value);
   else if (value >= -128 && value <= 127)
      emit(JVM.BIPUSH, value);
   else if (value >= -32768 && value <= 32767)
      emit (JVM.SIPUSH, value);
   else
      emit (JVM.LDC, value);
}
```

- Visitor method:

```
public void visit (IntLiteral x) {
   emitICONST(Integer.parseInt(x.Lexeme);
}
```

We will use similar mechanisms for float and bool literals.

Arithmetic Expressions E_1 +<int> E_2

Code template:

[[E_1 +<*int*> E_2]]:

[[E_1]]
[[E_2]]
emit("iadd")

result of [[E_2]]
result of [[E_1]]
...

Evaluation stack before iadd is executed

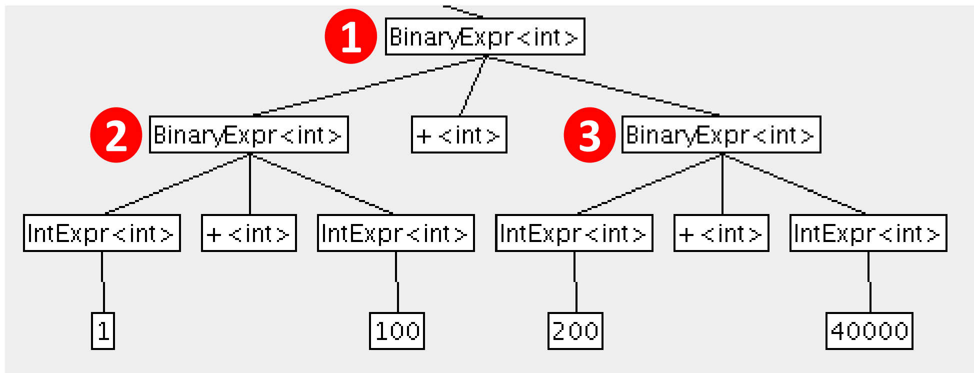
Visitor Method:

```
public void visit (BinaryExpr x) {
  x.lAST.accept (this); // emit code for left operand
  x.rAST.accept (this); // emit code for right operand
  ...
  else if ((x.oAST.type.Tequal(StdEnvironment.intType) && Op.equals("+"))
    emit(JVM.IADD);
  ...
}
```

Other arithmetic operators for int and float handled similarly (see following slides).

Expression Example: 1 + 100 + (200 + 40000)

- AST:



- The nodes visited in post-order ("iadd")...
- Emitted code:

```
iconst_1
bipush 100
iadd 2
sipush 200
ldc 40000
iadd 3
iadd 1
```

For the above AST, the code template from the previous slide is applied 3 times, once at each BinaryExpr node.

Boolean Expressions: E_1 && E_2

- Code Template

[[E_1 && E_2]] :

```
        [[ E1 ]]
        ifeq L1
        [[ E2 ]]
        ifeq L1
        iconst_1                    ; exit with ``true''
        goto L2
L1:
        iconst_0                    ; exit with ``false''
L2:
```

Code must respect the short-circuit evaluation rule (see next slide).

How to test Short-Circuit Evaluation

- Example:

```
bool f() {
   putBool(false);
   return false;
}

void main() {
  bool f;
  f = false && f();
}
```

- Wrong semantics if ``false'' is printed.
- **Reason**: evaluation of **&&** proceeds left-to-right and stops as soon as the expression is known to be false (short-circuit evaluation).

```
false && something  =  false
true || something = true
```

No need to evaluate ``something''.

- Note: short-circuit evaluation in C and MiniC is **mandatory** with the logical **and** (**&&**) and logical **or** (**||**) operators.

Assignment Statement i = E

- Assumptions for this example:
  - Variable i is of type int
  - Its local variable index is 1
- Code Template:

[[i = E]]:

```
[[ E ]]
istore_1
```

if (E) S_1 else S_2

- Code Template:

[[if (E) S_1 else S_2]]:

```
        [[ E ]]
        ifeq L1
        [[ S1 ]]              ; code for ``then'' branch
        goto L2
L1:
        [[ S2 ]]              ; code for ``else'' branch
L2:
```

- If the **else** branch is not present, then the code shown in blue need not be generated.

while (E) S

- Code Template:

  [[while (E) S]]:

```
L1:
    [[ E ]]
    ifeq L2
    [[ S ]]
    goto L1
L2:
```

- Also works if S is empty!

More on templates...

- The remaining templates will be provided to you with Assignment 5.
- Code generation for the JVM is then straight-forward.
- You have already done the main work in the earlier phases!

Slide 424…

- Here we are, after 423 slides on
  - Lexical Analysis,
  - Syntax Analysis,
  - Semantic Analysis,
  - Java Exceptions,
  - the Visitor Design Pattern,
  - Regular expressions, NFAs, DFAs, conversion algorithms between them,
  - Context-free grammars, concrete syntax, abstract syntax, LL(1) recursive descent (RD) parsers
  - Parser generators, attribute grammars
  - JVM architecture and code generation for the JVM
  - Programming language implementations
  - 2 Assignments & 2 Homework (not mandatory)

Example 1: Class MiniC.lang.System

```
package lang;

import java.io.*;
import java.util.StringTokenizer;

public class System {

  private static BufferedReader reader = new BufferedReader(
    new InputStreamReader(java.lang.System.in));

  public final static int getInt() {
    try {
      java.lang.System.out.print("Please enter an integer value: ");
      String s = reader.readLine();
      StringTokenizer st = new StringTokenizer(s);
      int i = Integer.parseInt(st.nextToken());
      java.lang.System.out.println("You have entered " + i + ".");
      return i;
    } catch(java.io.IOException e) {
      java.lang.System.out.println("Caught IOException " +
         e.getMessage());
      java.lang.System.exit(1);
      return -1;
    }
  }

  public final static void putInt(int i) {
     java.lang.System.out.print(i);
  }
  ...
}
```

- BufferedReader reads ahead and copies a chunk of characters from a file into an internal buffer of the JVM.
- Following readline() method calls can read from the buffer rather than accessing the underlying OS.
- Cost of a system call is higher than reading from the internal buffer.

- The StringTokenizer splits a string into tokens.
- Tokens are sub-strings delimited by one of " \t\n\r\f".
- These tokenization is conceptually much simpler than the regular expression mechanism we studied with our lexical analyzer.

Boolean Literals

- Code Template: **[[*BoolLiteral*]]:** emitBCONST(BoolLiteral.value)
  - Booleans represented as int values with the JVM.

```
private void emitBCONST(boolean value) {
   if (value) {
       emit(JVM.ICONST_1); // true = 1 with the JVM
   } else {
       emit(JVM.ICONST_0); // false = 0
   }
}
```

- Visitor method:

```
public void visit (BoolLiteral x) {
   emitBCONST(x.Lexeme.equals("true"));
}
```

Floating-point Literals

- Code Template: **[[*FloatLiteral*]]:** emitFCONST(FloatLiteral.value)

```
private void emitFCONST(float value) {
   if(value == 0.0) {
      emit(JVM.FCONST_0);
   } else if(value == 1.0) {
      emit(JVM.FCONST_1);
   } else if(value == 2.0) {
      emit(JVM.FCONST_2);
   } else {
      emit(JVM.LDC, value);
   }
}
```

Function Declarations

- Visitor method:

```
public void visit(FunDecl x) {
  isMain = x.idAST.Lexeme.equals("main");
  if (isMain) {
    frame = new Frame(true);
    emit ("\n.method public static main([Ljava/lang/String;)V");
  } else {
    frame = new Frame(false);
    emit ("\n.method public " + x.idAST.Lexeme
          + getDescriptor(x));
    x.paramsAST.accept(this); // adjust local variable count.
  }
  ...
}
```

- A new frame object is created each time the FunDecl visitor is called.
- The frame object is used during processing of the function body.
  - Code will be provided.

Method Frame

```
public Frame (boolean isMain) { // method frame constructor
   this.isMain = isMain;
   LabelNr = -1;
   if (this.isMain)    // main is the only static method with MiniC:
      LocalVarNr = 1;  // Slot 0: argv[]; slot 1: mc$
   else                // All other methods are instance methods:
      LocalVarNr = 0;  // Slot 0: this pointer
}

public int getNewLabel() {
      return ++LabelNr;
}

public int getNewLocalVarIndex() {
      return ++LocalVarNr;
}
```

- Labels must be unique within a Java method. Method `getNewLabel()` allows the code generator to request a new label.
- Local variable slots managed by the method frame object, too.

Local Variable Declarations

- A new index variable has been provided in the Decl AST class:

```
public abstract class Decl extends AST {

    private boolean global;
    public int index;          // local variable slot

    public Decl (SourcePos pos) {
        super (pos);
        global = false;
        index = -1;
    }

    ...
}
```

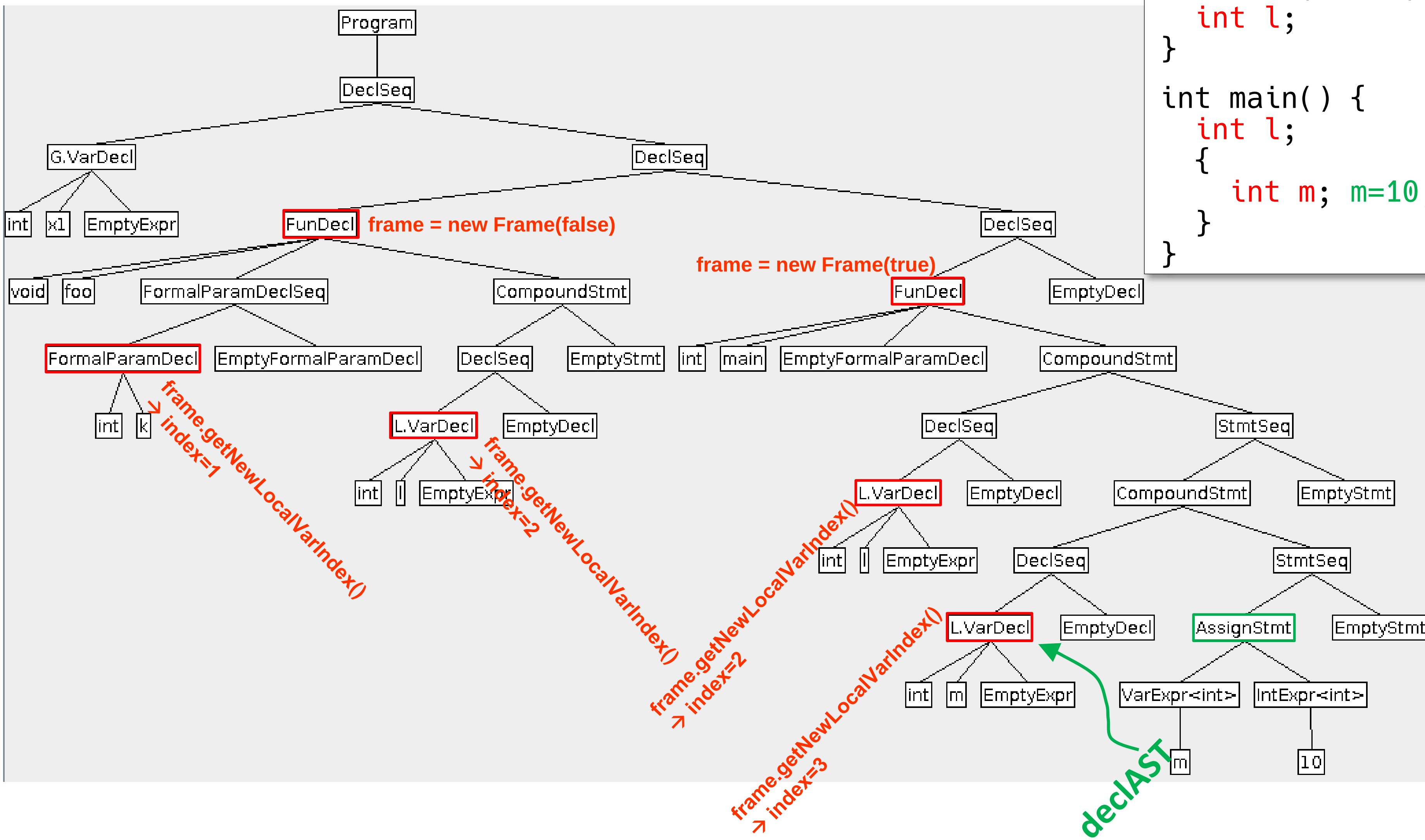
- This index variable is inherited by all declarations.
  - Variable declarations, formal parameter declarations.
- If a variable is local, the index denotes its local variable slot.
  - Call `frame.getNewLocalVarIndex()` to allocate a slot in the current method frame.

Allocation of local variable slots:

```
int x1;

void foo(int k) {
  int l;
}

int main() {
  int l;
  {
    int m; m=10;
  }
}
```



- Each time the code-generator Visitor reaches a FunDecl, a new Frame object is allocated.
- Each time a variable declaration is reached, a local variable index (=slot) is allocated from the current frame object. Variable accesses (load/store) will refer to this particular slot.

General assignment statement LHS = RHS

- Code Template:

[[LHS = RHS]]:

[[*LHS*]]

[[R*HS*]]

store instruction(s) appropriate for LHS

MiniC Example:

```
int a[10]; // index 1
int i = 1; // index 2
int j = 2; // index 3

a[i+1] = j + 5;
```

Bytecode for **a[i+1]=j+5:**

```
aload_1    ;; LHS array reference
iload_2
iconst_1       LHS array index value
iadd
iload_3
pipush 5       RHS value
iadd
iastore    ;; Store
```

while (E) S

- Code Template:

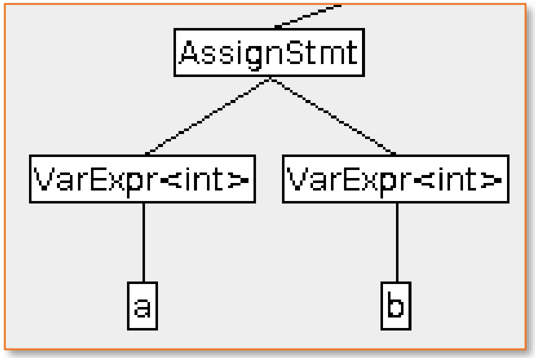
[[while (E) S]]:

```
L1:
    [[ E ]]
    ifeq L2
    [[ S ]]
    goto L1
L2:
```

- Also works if S is empty!
- `Break` statement can be implemented as ``goto L2''.
- `Continue` statement can be implemented as ``goto L1''.
- The labels are provided by the procedure's Frame object.
  - By calling `Frame.getNewLabel()`

lvalues (store) versus rvalues (load)

```
int a, b;
a = b;
```



- Create the appropriate **store** instruction in the visit(AssignStmt) method.
- Do not attempt this in visit(VarExpr), because "down" there we do not know whether the variable is an l-value (left-hand side, LHS) or an r-value (right-hand side, RHS).

[[return E]]:

- We assume that type coercion has already been done.
- Code template for return E<int> and return E<bool>:

```
[[ E ]]
ireturn
```

- Code template for return E<float>

```
[[ E ]]
freturn
```