COMP3131/9102: Programming Languages and Compilers

Jingling Xue

School of Computer Science and Engineering
The University of New South Wales
Sydney, NSW 2052, Australia

`http://www.cse.unsw.edu.au/~cs3131`

`http://www.cse.unsw.edu.au/~cs9102`

Copyright @2025, Jingling Xue

Week 8 (2nd Lecture): Java Byte Code Generation

1. Translation:
   - Expressions (including actual parameters)
   - Statements
   - Declarations (including formal parameters)
2. Allocating variable indices for local variables
3. Some special code generation issues:
   - lvalue (store) v.s rvalue (load)
   - assignment expressions such as “a = b[1+i] = 1”
   - Expression statements such as “1 + (a = 2);”
   - Short-circuit evaluations
   - break and continue
   - return
4. Generating Jasmin assembler directives
   - .limit stack
   - .limit locals
   - .var
   - .line

Code Generator as a Visitor Object

- Visitor (as an Object): implementing VC.ASTs.visitor
- Syntax-driven: traversing 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 defining the simple JVM used
Instruction.java: The class defining Jasmin instructions
Frame.java:       The class for info about labels, local
                  variable indices, etc. for a function
```

Code Template

- $[[X]]$: the code generated for construct X
- Code template: a specification of $[[X]]$ in terms of the codes for its syntactic components
- A code template specifies the translation of a construct independently of the context in which it is used
  - Compiled code always executes in some context
  - Optimisation is the art of captalising on context!
  - Lack of context $\Rightarrow$ fully general (i.e., slow) code
- Thus, inefficient code may be generated; it can be optimised later by the compiler backend

Our Translation Scheme: $X = 1 < 2$

```
int main() {
    int x;
    if (1 < 2)
      x = 10;
    else
      x = 20;
}
```

```
        iconst_1
        iconst_2
        if_icmplt L4
        iconst_0
        goto L5
L4:
        iconst_1
L5:
        ifeq L2
        bipush 10
        istore_2
        goto L3
L2:
        bipush 20
        istore_2
L3:
```

[[X]]

```
int main() {
    boolean x;
    x = (1 < 2);
}
```

```
        [[X]] // same code
        istore_2
```

More Optimized Code

```
int main() {
    int x;
    if (1 < 2)
      x = 10;
    else
      x = 20;
}
```

```
           iconst_1
           iconst_2
           if_icmpge L4
           bipush 10
           goto L5
L4:
           bipush 20
L5:
           istore_2
           goto L3
L2:
           bipush 20
           istore_2
```

- You are not required to generate such more optimised code
- The less optimised code given in the preceding slide can usually be further optimised into the code above by the compiler back end.
- Our translation scheme is simple (without focusing on producing efficient code.

Example 1: gcd.vc

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

int gcd(int a, int b) {
  if (b == 0)
    return a;
  else
  return gcd(b, a - (a/b) *b);
}

int main() {
  putIntLn(gcd(i, j));
  return 0;  // optional in VC or C/C++
}
```

Example 1: gcd.vc(Red Assumed by the VC Compiler)

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

  public gcd() { }  // the default constructor

  int gcd(int a, int b) {
    if (b == 0)
      return a;
    else
      return gcd(b, a - (a/b) *b);
  }

  void main(String argv[]) {
    gcd vc$;
    vc$ = new gcd();
    System.putIntLn(vc$.gcd(i, j));
    return;
  }
}
```

Example 1: gcd.vc (cont'd)

- int main() is assumed to be:

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

  - visitFuncDec: a return is always emitted just in case no "return expr" was present in the main of a VC program
  - visitReturnStmt: emit a RETURN rather than IRETURN even if a return statement, e.g., "return expr" is present in the main of a VC program
- All VC functions are assumed to be instance methods with the package access
- All global variables are assumed to be static field variables with the package access
- All built-in VC functions are static

Expressions

1. Literals
2. Variables (lvalues and rvalues)
3. Arithmetic expressions
4. Boolean expressions
5. Relational expressions
6. Assignment expressions
7. Call expressions (assignment spec)

Integer Literals

- CodeTemplate: $[[IntLiteral]]$: 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 Object visitIntLiteral(IntLiteral ast, Object o) {
    Frame frame = (Frame) o;
    emitICONST(Integer.parseInt(ast.spelling));
    ...
    return null;
  }
```

Floating-Point Literals

- CodeTemplate: $[[FloatLiteral]]$: 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);
}
```

- Visitor method:

```
public Object visitFloatLiteral(FloatLiteral ast, Object o) {
  Frame frame = (Frame) o;
  emitFCONST(Float.parseFloat(ast.spelling));
  ...
  return null;
}
```

Boolean Literals

- CodeTemplate: $[[BooleanLiteral]]$: emitBCONST(BooleanLiteral.value)

```
private void emitFCONST(boolean value) {
  if (value)
    emit(JVM.ICONST_1);
  else
    emit(JVM.ICONST_0);
}
```

- Visitor method:

```
public Object visitBooleanLiteral(BooleanLiteral ast, Object o) {
  Frame frame = (Frame) o;
  emitBCONST(ast.spelling.equals("true"));
  ...
  return null;
}
```

Arithmetic Expression $E_1 \ i+ \ E_2$

- Code template:

$$[[E_1 \ i+ \ E_2]]:$$

$$[[E_1]]$$
$$[[E_2]]$$
emit(”iadd”)

- Visitor Method:

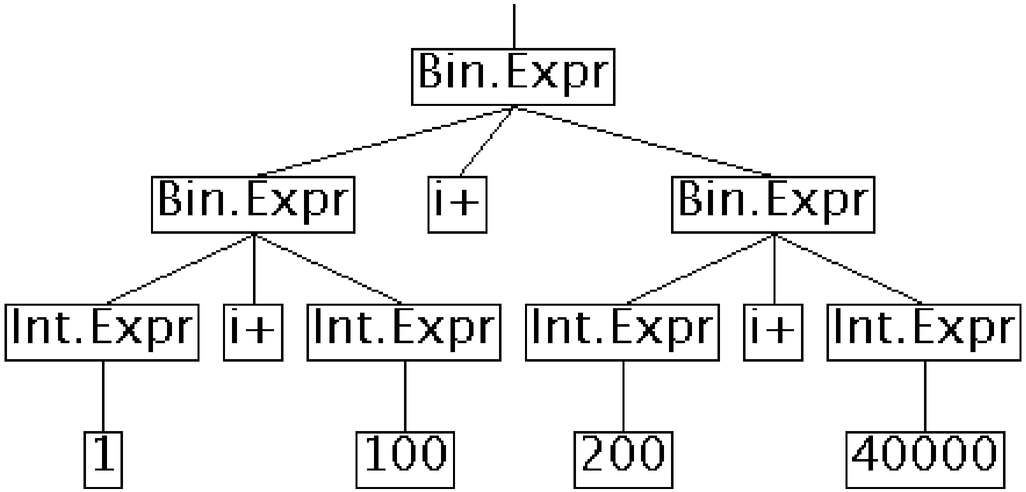
```
public Object visitBinaryExpr(BinaryExpr ast, Object o) {
    Frame frame = (Frame) o;
    String op = ast.O.spelling;

    ast.E1.visit(this, o);
    ast.E2.visit(this, o);
    ...
    else if (op.equals("i+")) {
        emit(JVM.IADD);
        ...
    }
    ...
```

- Other arithmetic operators (integral or real) handled similarly

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

- AST:



- The nodes visited in post-order per code template
- Code:

```
iconst_1
bipush 100
iadd
sipush 200
ldc 40000
iadd
iadd
```

visitFuncDecl: Frame Objects

- A new frame object created each time visitFuncDecl is called
- ```
  public Object visitFuncDecl(FuncDecl ast, Object o) {
    ...
    frame = new Frame(true) for main or new Frame(false) otherwise
    ...
  ```
- The frame object passed as the 2nd arg and available at all child nodes
- The constructor of the class Frame:

```
public Frame(boolean _main) {
  this._main = _main;
  label = 0;
  localVarIndex = 0;
  currentStackSize = 0;
  maximumStackSize = 0;
  conStack = new Stack<String>();
  brkStack = new Stack<String>();
  scopeStart = new Stack<String>();
  scopeEnd = new Stack<String>();
}
```

- Code will be provided
```

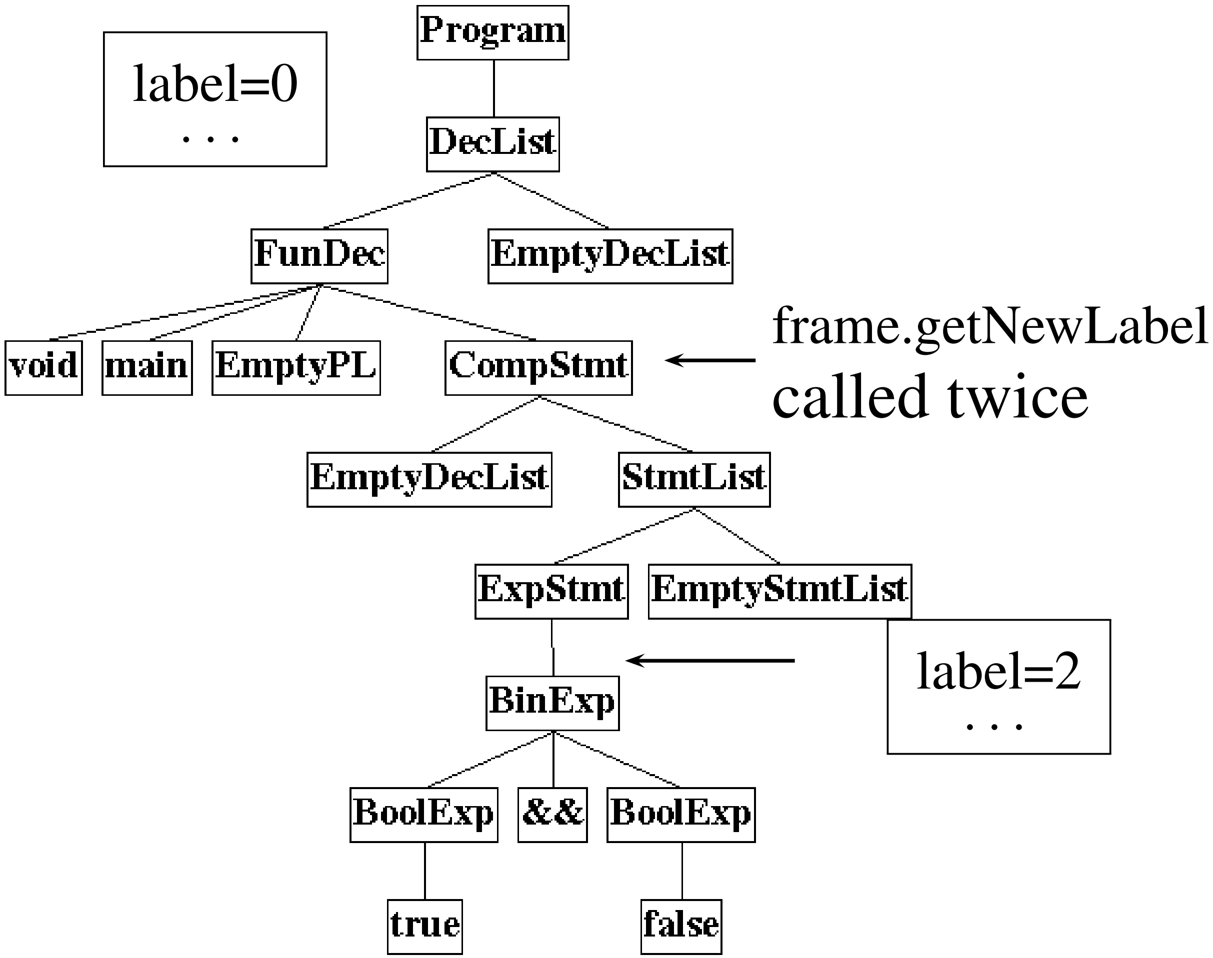
Boolean (or Logical) Expressions: $E_1 \&\& E_2$

$[[E_1]]$
ifeq Label1
$[[E_2]]$
ifeq Label1
iconst_1
goto Label2
Label1:
iconst_0
Label2:
	public Object visitBinaryExpr(BinaryExpr ast, Object o) {
	Frame frame = (Frame) o;
	...
	Label1 = frame.getNewLabel();
	Label2 = frame.getNewLabel();
	ast.E1.visit(this, o);
	emit(JVM.IFEQ, Label1);
	ast.E2.visit(this, o);
	emit(JVM.IFEQ, Label1);
	emit(JVM.ICONST_1);
	emit(JVM.GOTO, Label2);
	emit(Label1 + ":");
	emit(JVM.ICONST_0);
	emit(Label2 + ":");
	...

- Code must respect the short circuit evaluation rule
- || and ! dealt with similarly
- Better codes can be generated (Week 9 Tutorial)

Example 2: Boolean Expressions: true && false

```
        iconst_1
        ifeq L2
        iconst_0
        ifeq L2
        iconst_1
        goto L3
L2:
        iconst_0
L3:
```



- The Frame object created for main
- Passed to all the children of the main's FuncDecl node

Testing and Marking Short-Circuit Evaluation

- Example:

```
boolean f() {
  putBool(false);
  return false;
}
void main() {
  false && f();
}
```

- Wrong if ”false” is printed!

Relational Expressions: $E_1 \; i\!> \; E_2$

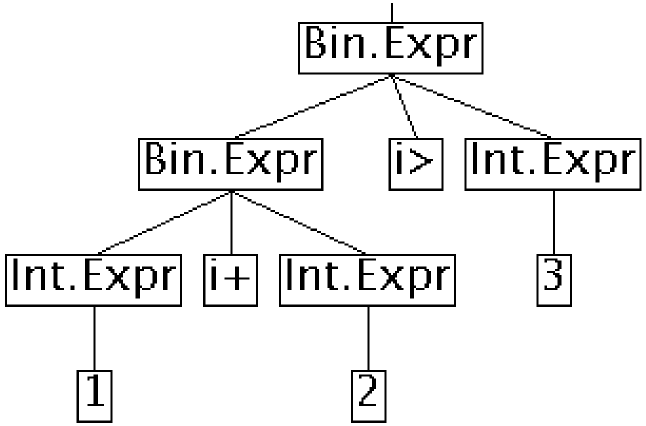
- Code Template:

```
        [[E1]]
        [[E2]]
        if_icmpgt L1
        iconst_0
        goto L2
L1:
        iconst_1
L2:
```

- Other relational operations on integer operands handled similarly

Example 3: Relational Expressions

- AST:



- Code – L0 and L1 generated in visitCompStmt

```
        iconst_1
        iconst_2
        iadd
        iconst_3
        if_icmpgt L2
        iconst_0
        goto L3
L2:
        iconst_1
L3:
```

Relational Expressions: $E_1\ f>\ E_2$

- Code Template:

```
        [[E1]]
        [[E2]]
        fcmpg
        ifgt L1
        iconst_0
        goto L2
L1:
        iconst_1
L2:
```

- if_fcmpgt is non-existent and is simulated by fcmpg and ifgt
- Other floating-point relational operators handled similarly

Assignment Expression: a = E

- Assumptions:

```
(1) a is int
(2) Its local variable index is 1
```

- Code Template:

$$[[E]]$$
istore_1

- The above code template breaks down for a = b = 1;

```
iconst_1
dup
istore_2 // the local var index for b is 2
istore_1
```

- Need to know the context in which b = 1 is used when the node for b=1 is visited
- How? a parent link is added to every AST node
- ast.parent is not $\cdots \Rightarrow$ dup

Assignment Expression: LHS = RHS

- Code Template:

$$[[LHS]]$$
$$[[RHS]]$$
appropriate store instruction

- Example:

```
VC:
int[] a = new int[10];  // index 1
int i = 1;              // index 2
int j = 2;              // index 3
a[i + 1] = j + 10;

Bytecode for a[i + 1] = j + 10:
        aload_1
        iload_2
        iconst_1
        iadd
        iload_3
        bipush 10
        iadd
        iastore
```

Statements

1. if
2. while — "for" left for you to work it out
3. break and continue
4. return
5. expression statement
6. compound statement

if (E) S1 else S2

- Code Template:

$$
\begin{array}{ll}
 & [[E]] \\
 & \text{ifeq L1} \\
 & [[S1]] \\
 & \text{goto L2} \\
\text{L1:} & \\
 & [[S2]] \\
\text{L2:} &
\end{array}
$$

- Works even when either S1 or S2 or both are empty
- In the AST, if (E) S1 without the **else** is represented as

```
       IfStmt
     /   |   \
    E    S1   EmptyStmt
```

Those instructions in blue need not be generated.

while (E) S

- Code Template:

Push the continue label L1 to conStack
Push the break label L2 to brkStack

L1:

$[[E]]$
ifeq L2
$[[S]]$
goto L1

L2:

Pop the continue label L1 from conStack
Pop the break label L2 from brkStack

- Also works when S is empty

break and continue

- Code template for break:

```
goto the label marking the inst following the while
```

- Code template for continue:

```
goto the label marking the first inst of the while
```

return E

- Assumption: type coercion has been done.
- Code Template: return E:int and return E:Boolean

$$[[E]]$$
ireturn

- Code Template: return E:float

$$[[E]]$$
freturn

Expression Statement: E;

- Code Template:

  $[[E]]$
  pop if it has a value left on the stack

- Examples:

```
1;        ---> pop
1 + 2;    ---> pop
f(1,2)    ---> pop if the return type is not void
a = 1;    ---> no pop
;         ---> no pop
```

Compound Statememts

- Code template:

```
Push the label marking the beginning of scope to scopeStart
Push the label marking the end of scope to scopeEnd
   ...
   [[DL]] // no code;
   [[SL]]
Pop the scopeStart label
Pop the scopeEnd label
```

- Code will be provided

Global Variable Declarations

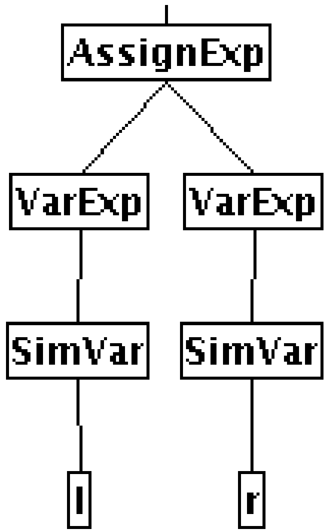
- Provided for you (but only for scalar variables)
  - Generate .field declarations
  - Geneate the class initialiser `<clinit>`
- You need to add the initialisations for arrays
- All initialisers for global variables are assumed to be constant expressions as in C, although this was not checked in Assignment 4.

Local Variable Declarations

- Instance field index available in VC.ASTs.Decl.java
- Call frame.getNewIndex() to allocate indices consecutively for formal parameters and local variables:
  - For a function (treated as an instance method), 0 is allocated to this
  - For main (a static method), 0 is allocated argv and 1 to the implicitly declared variable vc$

lvalues (store) v.s rvalues (load)

- Let visitSimpleVar do nothing (because we do not know by looking at this node whether the variable is a lvalue or rvalue)
- Generate an appropriate load or store in visitAssignExpr
- Consider l = r (store for l and load for r):



Generating Jasmin Directives

- .limit locals
- .limit stack
- .var
- .line

.limit locals XXX

- Generated at the end of processing a function
- XXX is the current value of frame.getNewIndex()

.var

- Syntax:

```
.var var-index is name type-desc scopeStart-label scopeEnd-label
```

- Generated when a var or formal para decl is processed
- var-index, name and type are extracted from the Decl node
- The scopeStart and scopeEnd labels from scopeStart and scopeEnd stacks (Slide 516)

.line XXX

- Source line where the instructions between this .line and the next are translated from
- Optional (you should leave it at the very end)
- Maintain a current line
- Generate a .line if the next construct is from a different line

.limit stack XXX

- XXX is the maximum depth of the operand stack
- Calculating the value by simulating the execution of the byte code generated incrementally
- Example:

```
iconst_1    frame.push()
iconst_2    frame.push()
iadd        frame.pop()
iconst_1    frame.push()
iconst_2    frame.push()
iconst_3    frame.push()
iadd        frame.pop()
iadd        frame.pop()
astore_1    frame.pop()
...
```

Some Language Issues

- Java byte code requires that
  - all variables be initialised
  - all method be terminated by a return
- Both are not enforced in the VC language
- All test cases used for marking Assignment 5 will satisfy these two restrictions.

Reading

- Chapter 7 of the on-line JVM Spec (compiling Java)
- §8.4 (Red Dragon) or §6.6.2 of Purple Dragon (for short-circuit evaluations)

Next Class: Code Generation