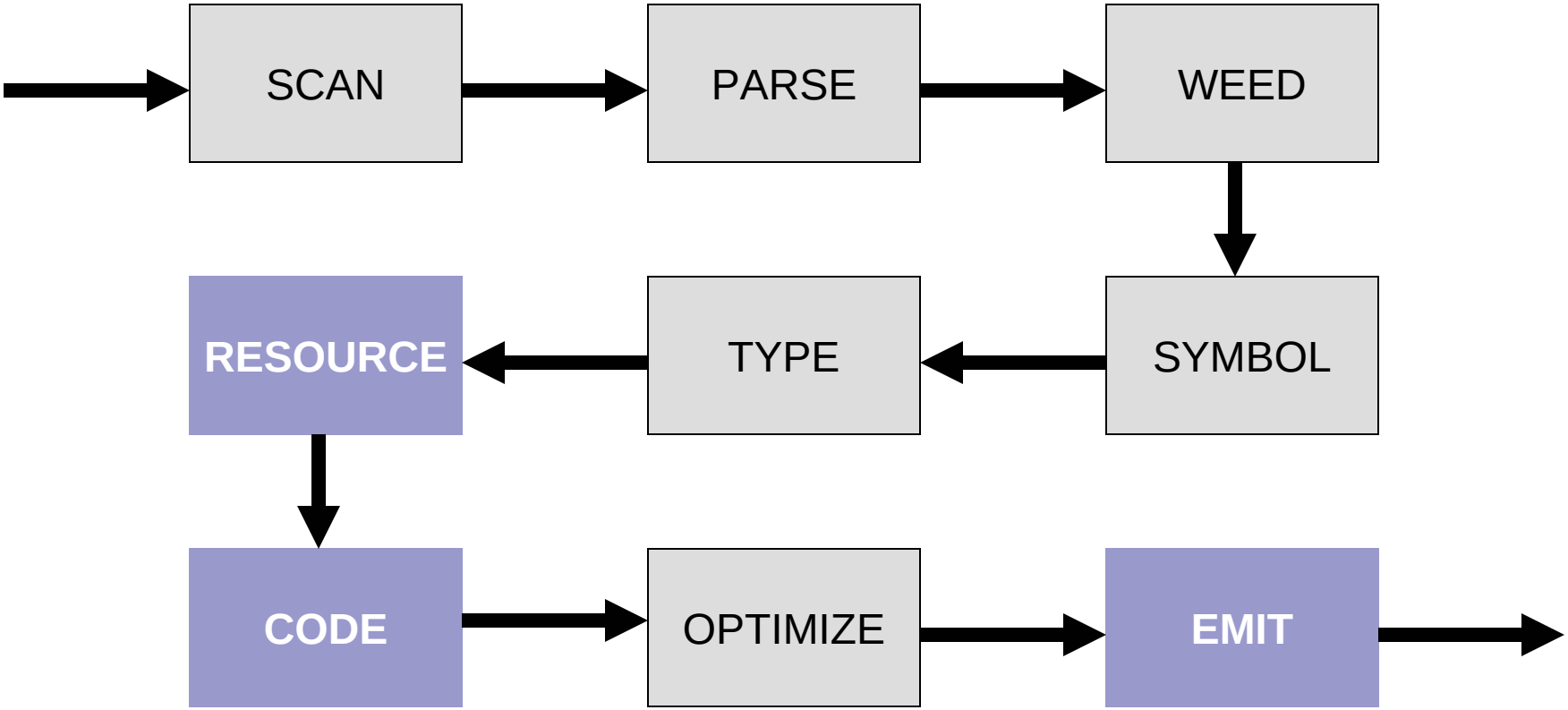
CIS 706
Translator Design I

Code Generation

Compiler Architecture



Code Generation Phases

- Translating classes and fields
- Translating methods
  - computing *resources* such as layouts, offsets, labels, registers, and dimensions;
  - generating an internal representation of machine codes for statements and expressions;
  - optimizing the code (ignored for now); and
- emitting the code to files in binary format.

ByteCodeGenerator

Overview Package **Class** Tree Deprecated Index Help

PREV CLASS NEXT CLASS | FRAMES NO FRAMES
SUMMARY: NESTED | FIELD | CONSTR | METHOD | DETAIL: FIELD | CONSTR | METHOD

sjc.codegen

Class ByteCodeGenerator

```
java.lang.Object
  └ sjc.codegen.ByteCodeGenerator
```

```
public class ByteCodeGenerator
extends java.lang.Object
```

This class is used to translate a StaticJava `CompilationUnit` to `ClassByteCodes` that represent a Java 1.5 class files.

Author:
Robby

Nested Class Summary

static class	`ByteCodeGenerator.Error` This class is used to signal an error in the process of generating bytecode.

Method Summary

static `ClassByteCodes`	`generate(org.eclipse.jdt.core.dom.CompilationUnit cu, SymbolTable st, TypeTable tt)` Generates a `ClassByteCodes` that represents the class files for the given StaticJava `CompilationUnit` with the given `SymbolTable` and `TypeTable`.

Methods inherited from class java.lang.Object

`equals, getClass, hashCode, notify, notifyAll, toString, wait, wait, wait`

ByteCodeGenerator.Visitor

```
protected static class Visitor extends ASTVisitor {
  protected @NonNull ClassWriter cw;
  protected @PossiblyNull FieldVisitor fv;
  protected @PossiblyNull MethodVisitor mv;
  public @PossiblyNull String mainClassName;
  public @PossiblyNull byte[] mainClassBytes;
  protected @NonNullElements @ReadOnlyElements Map<ASTNode, Object> symbolMap;
  protected @NonNullElements @ReadOnlyElements Map<ASTNode, Type> typeMap;
  protected @NonNullElements @ReadOnlyElements Map<Object, Pair<Type,
    List<Type>>> methodTypeMap;
  protected @NonNull @NonNullElementsOnly List<Pair<String, Type>>
    localNamesTypes = new ArrayList<Pair<String, Type>>();
  protected @NonNull @NonNullElementsOnly Map<String, Integer> localIndexMap =
    new HashMap<String, Integer>();

 protected Visitor(@NonNull SymbolTable st, @NonNull TypeTable tt) {
   assert st != null && tt != null;
   symbolMap = st.symbolMap;
   typeMap = tt.typeMap;
   methodTypeMap = tt.methodTypeMap;
 } … }
```

Type Declaration

```
public boolean visit(TypeDeclaration node) {
  mainClassName = node.getName().getIdentifier();
  cw = new ClassWriter(true, false);
  cw.visit(V1_5, ACC_PUBLIC + ACC_SUPER, mainClassName, null,
           "java/lang/Object", null);
  cw.visitSource(null, null);
  generateConstructor(mainClassName);
  for (Object o : node.bodyDeclarations()) {
    ((ASTNode) o).accept(this);
  }
  cw.visitEnd();
  mainClassBytes = cw.toByteArray();
  cw = null;
  return false;
}
```

Field Declaration

```
public boolean visit(FieldDeclaration node) {
  int modifiers = convertModifiers(node, node.modifiers());
  VariableDeclarationFragment vdf = (VariableDeclarationFragment) node
                                    .fragments().get(0);
  cw.visitField(modifiers, vdf.getName().getIdentifier(),
                convertType(typeMap.get(node)), null, null).visitEnd();
  return false;
}
```

Method Declaration

```
public boolean visit(MethodDeclaration node) {
  String methodName = node.getName().getIdentifier();
  int modifiers = convertModifiers(node, node.modifiers());
  Pair<Type, List<Type>> methodType = methodTypeMap.get(node);
  String methodDesc = getMethodDescriptor(methodType.first, methodType.second);
  mv = cw.visitMethod(modifiers, methodName, methodDesc, null, null);
  mv.visitCode();
  Label initLabel = new Label();
  mv.visitLabel(initLabel);
  buildLocalIndexTable(node);
  Statement lastStatement = null;
  for (Object o : node.getBody().statements()) {
    if (!(o instanceof VariableDeclarationStatement)) {
      ((ASTNode) o).accept(this);
    }
    lastStatement = (Statement) o;
  }

  …
}
```

Method Declaration

```
public boolean visit(MethodDeclaration node) {
  …
  // TODO: this is a hack, we need an analysis to see whether we need
  // a return statement or not.
  if (!(lastStatement instanceof ReturnStatement)) {
    if (methodType.first instanceof VoidType) { mv.visitInsn(RETURN); }
  }
  Label endLabel = new Label();
  mv.visitLabel(endLabel);
  int i = 0;
  for (Pair<String, Type> p : localNamesTypes) {
    mv.visitLocalVariable(p.first, convertType(p.second), null,
                          initLabel, endLabel, i);
    i++;
  }
  mv.visitMaxs(0, 0);
  mv.visitEnd();
  localNamesTypes.clear();
  localIndexMap.clear();
  return false;
}
```

Resources in Java

Include

- offsets for locals and formals;
- stack limit and locals limit for methods; and
- labels for control structures.

These are values that cannot be computed locally.

- We must perform a global traversal of the parse trees.

Computing Locals Limit

```
public class Example {
  public Example() { super(); }
  public void method(int p, int q, Example r) {
    int x, y;
    { int z;
      z = 87;
    }
    { boolean a;
      Example x2;
      { boolean b;
        int z;
        b = true;
      }
      { int y2;
        y2 = x = 0;
      }
    }
  }
}
```

1 2 3 4 5 6 6 7 8 9 8

Keep track of number of declarations in block

Maximum number of active declarations plus one (for receiver) is locals limit

Stack Limits

- Computing the stack limit is much harder.
  - It is the maximum height of the stack during the evaluation of an expression in the method.

- This requires detailed knowledge of:
  - the code that is generated; and
  - the virtual machine.

Computing Stack Limits

```
public void method() {
   int x, y;
   x = 12;
   y = 87;
   x = 2 * (x + y * (x - y));
}
```

Computing Stack Limits

```
public method()V
 L0 (0)
  LINENUMBER 21 L0
  BIPUSH 12             // 1
  ISTORE 1              // 0
 L1 (3)
  LINENUMBER 22 L1
  BIPUSH 87             // 1
  ISTORE 2              // 0
 L2 (6)
  LINENUMBER 23 L2
  ICONST_2              // 1
  ILOAD 1               // 2
  ILOAD 2               // 3
  ILOAD 1               // 4
  ILOAD 2               // 5
  ISUB                  // 4
  IMUL                  // 3
  IADD                  // 2
  IMUL                  // 1
  ISTORE 1              // 0
 L3 (17)
  LINENUMBER 24 L3
  RETURN                // 0
 L4 (19)
  LOCALVARIABLE this LExample; L0 L4 0
  LOCALVARIABLE x I L1 L4 1
  LOCALVARIABLE y I L2 L4 2
  MAXSTACK = 5
  MAXLOCALS = 3
```

Labels for Control Structures

`if`:	1 label
`if-else`:	2 labels
`while`:	2 labels
`||` and `&&`:	1 label
`==`, `>`, `<`, `>=`, `<=`, and `!=`:	2 labels
`!`:	2 labels

Labels are generated consecutively, for each method and constructor

If-statement

The statement

```
if (e) S
```

has code template:

```
e
ifeq elseOrEnd
S
elseOrEnd:
```

```
public boolean visit(IfStatement node) {
  Label elseOrEndLabel = new Label();
  node.getExpression().accept(this);
  mv.visitJumpInsn(IFEQ, elseOrEndLabel);
  node.getThenStatement().accept(this);
  Block elseBlock = (Block)
node.getElseStatement();
  if (elseBlock == null) {
    mv.visitLabel(elseOrEndLabel);
  } else { … }
  return false;
}
```

If-else-statement

The statement

`if (e)` S_1 `else` S_2

has code template:

```
e
ifeq elseOrEnd
S1
goto end
elseOrEnd:
S2
end:
```

```
public boolean visit(IfStatement node) {
  Label elseOrEndLabel = new Label();
  node.getExpression().accept(this);
  mv.visitJumpInsn(IFEQ, elseOrEndLabel);
  node.getThenStatement().accept(this);
  Block elseBlock = (Block) node.getElseStatement();
  if (elseBlock == null) { … } else {
    Label endLabel = new Label();
    mv.visitJumpInsn(GOTO, endLabel);
    mv.visitLabel(elseOrEndLabel);
    elseBlock.accept(this);
    mv.visitLabel(endLabel);
  }
  return false; }
```

While-statement

The statement

```
while (e) S
```

has code template:

```
loop
e
ifeq end
S
goto loop
end:
```

```
public boolean visit(WhileStatement node) {
  Label loopLabel = new Label();
  Label endLabel = new Label();
  mv.visitLabel(loopLabel);
  node.getExpression().accept(this);
  mv.visitJumpInsn(IFEQ, endLabel);
  node.getBody().accept(this);
  mv.visitJumpInsn(GOTO, loopLabel);
  mv.visitLabel(endLabel);
  return false;
}
```

Assignment

The statement

```
x = e
```

has code template:

```
e
istore offset(x)
```

if *x* is int or boolean, otherwise:

```
e
astore offset(x)
```

if *x* is static field use

`putstatic` *C*.*x*: type

Assignment

```
public boolean visit(Assignment node) {
  node.getRightHandSide().accept(this);
  ASTNode lhsNode = node.getLeftHandSide();
  String varName = ((SimpleName) lhsNode).getIdentifier();
  Object lhsDecl = symbolMap.get(lhsNode);
  if (lhsDecl instanceof FieldDeclaration) {
    FieldDeclaration fd = (FieldDeclaration) lhsDecl;
    String className = ((TypeDeclaration) fd.getParent())
                              .getName().getIdentifier();
    mv.visitFieldInsn(PUTSTATIC, className, varName,
                      convertType(typeMap.get(fd)));
  } else {
    mv.visitVarInsn(ISTORE,
                    localIndexMap.get(varName).intValue());
  }
  return false;
}
```

Expression Statement

The statement *e* has code template:

e

if *e* is void/assign otherwise:

e
pop

```
public boolean visit(ExpressionStatement node) {
  Expression e = node.getExpression();
  e.accept(this);
  Type t = typeMap.get(e);
  if (t == null || t instanceof VoidType) {
    // skip
  } else { mv.visitInsn(POP); }
  return false;
}
```

Local Variable Expression

The expression has code template:

x

`iload` *offset*(*x*)

if *x* is int or boolean otherwise:

`aload` *offset*(*x*)

```
public boolean visit(SimpleName node) {
  ASTNode parent = node.getParent();
  if (parent instanceof Expression || parent instanceof Statement) {
    String varName = node.getIdentifier(); Object decl = symbolMap.get(node);
    if (decl instanceof FieldDeclaration) { … } else {
      Type t = typeMap.get(node);
      if (t instanceof IntType || t instanceof BooleanType) {
        mv.visitVarInsn(ILOAD, localIndexMap.get(varName).intValue());
      } else {
        mv.visitVarInsn(ALOAD, localIndexMap.get(varName).intValue());
      } } } return false; }
```

Static Field Expression

The expression has code template:

x

`getstatic` *C.x:* type

```
public boolean visit(SimpleName node) {
  ASTNode parent = node.getParent();
  if (parent instanceof Expression || parent instanceof Statement) {
    String varName = node.getIdentifier(); Object decl = symbolMap.get(node);
    if (decl instanceof FieldDeclaration) {
      FieldDeclaration fd = (FieldDeclaration) decl;
      String className = ((TypeDeclaration) fd.getParent())
                              .getName().getIdentifier();
      mv.visitFieldInsn(GETSTATIC, className, varName,
                        convertType(typeMap.get(fd)));
    } else { … } } return false; }
```

|| Expression

The expression

$e_1 \mid\mid e_2$

has code template:

e_1
dup
ifne true
pop
e_2
true:

```
public boolean visit(InfixExpression node) {
  InfixExpression.Operator op = node.getOperator();
  node.getLeftOperand().accept(this);
  if (op == InfixExpression.Operator.CONDITIONAL_AND { … }
  else if (op == InfixExpression.Operator.CONDITIONAL_OR) {
    mv.visitInsn(DUP);
    Label trueLabel = new Label();
    mv.visitJumpInsn(IFNE, trueLabel);
    mv.visitInsn(POP);
    node.getRightOperand().accept(this);
    mv.visitLabel(trueLabel);
  } …
  return false;
}
```

== Expression

The expression $e_1 == e_2$ has code template:

```
e1
e2
if_icmpeq then
iconst_0
goto end
then:
iconst_1
end:
```

if e_i is int or boolean

== Expression

```
public boolean visit(InfixExpression node) {
  InfixExpression.Operator op = node.getOperator();
  node.getLeftOperand().accept(this);
  … else {
    node.getRightOperand().accept(this);
    … else if (op == InfixExpression.Operator.EQUALS) {
      generateRelationalCode(IF_ICMPEQ);
    } } return false; }

protected void generateRelationalCode(int opcode) {
  Label thenLabel = new Label();
  Label endLabel = new Label();
  mv.visitJumpInsn(opcode, thenLabel);
  mv.visitInsn(ICONST_0);
  mv.visitJumpInsn(GOTO, endLabel);
  mv.visitLabel(thenLabel);
  mv.visitInsn(ICONST_1);
  mv.visitLabel(endLabel);
}
```

+ Expression

The statement has code template:

$e_1 + e_2$

e_1
e_2
`iadd`

```
public boolean visit(InfixExpression node) {
  InfixExpression.Operator op = node.getOperator();
  node.getLeftOperand().accept(this);
  … else {
    node.getRightOperand().accept(this);
    if (op == InfixExpression.Operator.PLUS) {
      mv.visitInsn(IADD);
    } …
  }
  return false;
}
```

! Expression

The expression

```
! e
```

has code template:

```
e
ifeq false
iconst_0
goto end
false:
iconst_1
end:
```

```
public boolean visit(PrefixExpression node) {
  node.getOperand().accept(this);
  PrefixExpression.Operator op = node.getOperator();
  … else if (op == PrefixExpression.Operator.NOT) {
    Label falseLabel = new Label();
    Label endLabel = new Label();
    mv.visitJumpInsn(IFEQ, falseLabel);
    mv.visitInsn(ICONST_0);
    mv.visitJumpInsn(GOTO, endLabel);
    mv.visitLabel(falseLabel);
    mv.visitInsn(ICONST_1);
    mv.visitLabel(endLabel);
  }
  return false; }
```

Static Invoke Expression

The expression has code template:

$C.m(e_1, \ldots, e_n)$

e_1

…

e_n

`invokestatic` *`sig`*`(`*`C`*`, m)`

Where :

C is the declaring class of *m*

`sig`(*C*, *m*) is the signature of *m* in *C*

Static Invoke Expression

```
public boolean visit(MethodInvocation node) {
  for (Object o : node.arguments()) { ((ASTNode) o).accept(this); }
  Object o = symbolMap.get(node);
  if (o instanceof MethodDeclaration) {
    MethodDeclaration md = (MethodDeclaration) o;
    String className = ((TypeDeclaration) md.getParent())
                            .getName().getIdentifier();
    String methodName = node.getName().getIdentifier();
    Pair<Type, List<Type>> p = methodTypeMap.get(md);
    mv.visitMethodInsn(INVOKESTATIC, className, methodName,
                       getMethodDescriptor(p.first, p.second));
  } else if (o instanceof Method) {
    Method m = (Method) o;
    String className = m.getDeclaringClass().getName().replace('.', '/');
    String methodName = m.getName();
    Pair<Type, List<Type>> p = methodTypeMap.get(m);
    mv.visitMethodInsn(INVOKESTATIC, className, methodName,
                       getMethodDescriptor(p.first, p.second));
  }
  return false; }
```

Class File (in hex)

```
cafe babe 0003 002d 001a 0100 064c 5472
6565 3b07 0010 0900 0200 0501 0015 284c
6a61 7661 2f6c 616e 672f 4f62 6a65 6374
3b29 560c 0018 0001 0100 0654 7265 652e
6a01 000a 536f 7572 6365 4669 6c65 0100
0443 6f64 6507 000d 0c00 0e00 1209 0002
0017 0100 2128 4c6a 6176 612f 6c61 6e67
2f4f 626a 6563 743b 4c54 7265 653b 4c54
7265 653b 2956 0100 106a 6176 612f 6c61
6e67 2f4f 626a 6563 7401 0005 7661 6c75
650c 0011 0019 0100 0454 7265 6501 0006
3c69 6e69 743e 0100 124c 6a61 7661 2f6c
616e 672f 4f62 6a65 6374 3b0a 0009 000f
0100 0873 6574 5661 6c75 6509 0002 000a
0100 046c 6566 740c 0016 0001 0100 0572
6967 6874 0100 0328 2956 0021 0002 0009
0000 0003 0006 000e 0012 0000 0006 0016
0001 0000 0006 0018 0001 0000 0002 0001
0011 000c 0001 0008 0000 0020 0003 0004
0000 0014 2ab7 0013 2a2b b500 152a 2cb5
000b 2a2d b500 03b1 0000 0000 0001 0014
0004 0001 0008 0000 0012 0003 0002 0000
0006 2a2b b500 15b1 0000 0000 0001 0007
0000 0002 0006
```

Code Generator Testing

- Design a convention for linking templates
- Local arguments that each code template is correct and that it meets the convention
- Ensure coverage of all of the code template variations in testing