Overview

- Java Compilation and Intermediate Representation
  - Bytecode
- JoeQ Framework
  - Quads: Instruction set used in JoeQ
  - JoeQ constructs
  - Writing analysis in JoeQ
- HW2
Typical Compiler Infrastructure

Lexing
Parsing
Type checking

Front-end

Machine-independent optimizations

Middle-end

Machine-dependent optimizations

Back-end
Java

Lexing
Parsing
Type checking

Java compiler (javac)

Front-end

Machine-independent optimizations

Middle-end

Machine-dependent optimizations

JVM (java)

Back-end
Java Source Code

- Input to the Java front-end
- A very “rich” representation
  - Human readable and writable
  - Hard to analyze (from the perspective of a computer)
- Many high-level concepts which do not direct hardware counterparts
  - Classes, generics, virtual function calls, exceptions, structured control flow, locks, etc.
Abstract Syntax Tree Representation

- Abstract syntax trees generated by parser that can be analyzed before code generation
  - Attribution (com.sun.tools.javac.comp.Attr)
  - Dataflow (com.sun.tools.javac.comp.Flow)
    - Liveness, exception checks, assignment/unassignment, local variable capture, etc.
  - Desugaring
- Analyses implemented using “visitor” pattern
Java Bytecode

- Machine-independent intermediate representation (.class files)
- Coarse program structure is still maintained
  - One file per class
  - A section per method or field
- Each method has a bytecode sequence for its implementation
- Remains high-level
  - Still have virtual methods, locks, etc.
Bytecode representation

● Each bytecode instruction uses one byte
  ○ Instructions may have additional operands, stored immediately after the instruction
● Verification
● Stack machine model
  ○ All intermediate values stored on a stack
What is a stack machine model?

- Each instruction pushes or pops values onto a stack
- Working example: $x = y + 10$
What is a stack machine model?

- Each instruction pushes or pops values onto a stack
- Working example: $x = y + 10$

```
push y
```

```
y
```
What is a stack machine model?

- Each instruction pushes or pops values onto a stack
- Working example: \( x = y + 10 \)

```
Instructions
→ push y
→ push 10
```

```
Stack
10
y
```
What is a stack machine model?

- Each instruction pushes or pops values onto a stack
- Working example: \( x = y + 10 \)

**add** - implicitly pops the top 2 operands and pushes the result back on the stack
What is a stack machine model?

- Each instruction pushes or pops values onto a stack
- Working example: \( x = y + 10 \)

** Instructions **

- push y
- push 10
- add
- pop x

** Stack **

- y+10

** pop ** - top of stack is popped and the value assigned to \( x \)
Bytecode Instructions

- Each instruction is prefixed by the types of operands
- `iload_1`
  - Load the first parameter or local variable and push it on the stack
- `bipush <n>`
  - Push byte constant \( n \) onto the stack as an integer value
- `iadd`
  - Add the top two values on the stack, and push the result back onto the stack
- `istore_2`
  - Pop the stack and store its value into the second param/local
ExprTest.test(int a)

class ExprTest {
    int test(int a) {
        int b, c, d, e, f;
        c = a + 10;
        f = a + c;
        if (f > 2) {
            f = f - c;
        }
        return (f);
    }
}

> javac ExprTest.java
> javap -c ExprTest

class ExprTest extends java.lang.Object {
    ExprTest();
    int test(int);
}

Method ExprTest():
    0 aload_0
    ...
    4 return

Method int test(int):
    0 iload_1
    ...
    24 ireturn
Method ExprTest():

0  aload_0
   // load address 'this' and push it onto the stack

1  invokespecial #1 <Method java.lang.Object()>
   // invokes base class methods. #1 is constructor

4  return
class ExprTest {
    int test (int a) {
        int b, c, d, e, f;
        c = a + 10;
        f = a + c;
        if (f > 2) {
            f = f - c;
        }
        return (f);
    }
}

Method int test(int):
    0 iload_1
    1 bipush 10
    3 iadd
    4 istore_3
    5 iload_1
    6 iload_3
    7 iadd
    8 istore_6
    10 iload_6
    12 icode
    13 if icmple 22
    14 iload_6
    16 iload_3
    18 iload_6
    19 isub
    20 istore_6
    22 iload_6
    24 ireturn
Static vs. Dynamic Optimization

- **Static optimization**
  - Dataflow analysis

- **Adaptive optimization**
  - Incorporate information available only at run-time
  - Interpreted bytecode vs. compiled native code
  - Determining hot-spots in code execution to be compiled to native code
JoeQ

- Compiler framework for analyzing and optimizing Java bytecode
  - Developed by John Whaley and others
  - Implemented in Java
  - Research project infrastructure: 10+ papers rely on JoeQ
- Also see: http://joeq.sourceforge.net
- Etymology
  - "Jyo-kyu-" like the name "Joe" and the letter "Q"
Java Elements in JoeQ

High level representation: JoeQ has class for various components of Java .class files, in `joeq.Class` package

- `jq_Type`
- `jq_Class`
- `jq_Field`
- `jq_Method`
- ...
JoeQ Intermediate Representation

- JoeQ uses a lower-level representation called a *Quad*
- JoeQ translates bytecodes into *Quads*
- A *Quad* has one operator and up to four operands: four-address instructions
- Stanford version of TAC (Three address code)
  - Example: $t1 := t2 + t3$, $t1 = 2 * t3$
class ExprTest {
    int test (int a) {
        int b, c, d, e, f;
        c = a + 10;
        f = a + c;
        if (f > 2) {
            f = f - c;
        }
        return (f);
    }
}
ExprTest.ExprTest()

Code:
0: aload_0
   // load address 'this' and push it onto the stack
1: invokespecial #1
   // invokes base class methods. #1 is constructor
2: return

B0 (ENTRY) (in: <none>, out: BB2)

BB2 (in: BB0 (ENTRY), out: BB1 (EXIT))
2  NULL_CHECK T-1 <g>,                         R0 ExprTest
1  INVOKEVIRTUAL java.lang.Object.<init> ()V,    (R0 ExprTest)
3  RETURN_V

BB1 (EXIT) (in: BB2, out: <none>)
Method int test(int):
  0 iload_1
  1 bipush 10
  3 iadd
  4 istore_3
  5 iload_1
  6 iload_3
  7 iadd
  8 istore_6
 10 iload 6
12 icall
13 iadd
16 iload 6
18 iload_3
19 isub
20 iload 6
22 iload 6
24 ireturn

Bytecode

BB0 (ENTRY)  (in: <none>, out: BB2)
BB2  (in: BB0 (ENTRY), out: BB3, BB4)
  1 ADD_I T0 int, R1 int, IConst: 10
  2 MOVE_I R3 int, T0 int
  3 ADD_I T0 int, R1 int, R3 int
  4 MOVE_I R6 int, T0 int
  5 IFCMP_I R6 int, IConst: 2, LE, BB4
BB3  (in: BB2, out: BB4)
  6 SUB_I T0 int, R6 int, R3 int
  7 MOVE_I R6 int, T0 int
BB4  (in: BB2, BB3, out: BB1 (EXIT))
  8 RETURN_I R6 int
BB1 (EXIT)  (in: BB4, out: <none>)

Quads
Quads \((joeq.\text{Compiler}.\text{Quad}.\text{Quad})\)

Register machine model, not stack machine model

- All temporary data stored in registers
- Closer to RISC machine instructions than a stack model
  - JoeQ is a lower-level IR than Java bytecode
- More conducive to program optimization than the stack architecture
Register Machine Model

- This mode assumes an unbounded number of pseudo-registers
- Pseudo-registers hold local variables of a method and temporary variables
- Data must first be loaded into pseudo registers before operating
class ExprTest {
    int test (int a) {
        int b, c, d, e, f;
        c = a + 10;
        f = a + c;
        if (f > 2) {
            f = f - c;
        }
        return (f);
    }
}

BB0 (ENTRY) (in: <none>, out: BB2)
BB2 (in: BB0 (ENTRY), out: BB3, BB4)
1 ADD_I    T0 int, R1 int, IConst: 10
2 MOVE_I   R3 int, T0 int
3 ADD_I    T0 int, R1 int, R3 int
4 MOVE_I   R6 int, T0 int
5 IFCMP_I  R6 int, IConst: 2, LE, BB4
BB3 (in: BB2, out: BB4)
6 SUB_I    T0 int, R6 int, R3 int
7 MOVE_I   R6 int, T0 int
BB4 (in: BB2, BB3, out: BB1 (EXIT))
8 RETURN_I R6 int
BB1 (EXIT) (in: BB4, out: <none>)
Quad Operators and Operands

- quad.getOperator() gives operator for quad
- getAllOperands() ... gives a list of all operands for a quad
- More meaningful methods such as getDest() to get destination register for a move instruction
  - Hierarchically ordered operators
  - Different operand types

BB2  (in: BB0 (ENTRY), out: BB3, BB4)
1  ADD_I       T0 int, R1 int, IConst: 10
2  MOVE_I      R3 int, T0 int
3  ADD_I       T0 int, R1 int, R3 int
4  MOVE_I      R6 int, T0 int
5  IFCMP_I     R6 int, IConst: 2, LE, BB4
JoeQ Operators (joeq.Compiler.Quad.Operators)

- Operator.Move
- Operator.Invoke
- Have suffixes indicating return type
  - For example, ADD_I adds two ints
  - Suffixes I, L, F, D, A, V refer to int, long, float, double, reference, void

```
BB0 (ENTRY)   (in: <none>, out: BB2)
BB2           (in: BB0 (ENTRY), out: BB3, BB4)
  1  ADD_I  T0 int,     R1 int, IConst: 10
  2  MOVE_I R3 int,     T0 int
  3  ADD_I  T0 int,     R1 int, R3 int
  4  MOVE_I R6 int,     T0 int
  5  IFCMP_I R6 int,     IConst: 2, LE, BB4
BB3           (in: BB2, out: BB4)
  6  SUB_I  T0 int,     R6 int, R3 int
  7  MOVE_I R6 int,     T0 int
BB4           (in: BB2, BB3, out: BB1 (EXIT))
  8  RETURN_I R6 int
BB1 (EXIT)    (in: BB4, out: <none>)
```
JoeQ Runtime Check Operators

- Runtime checks are explicit quads
  - Not implicit as in bytecodes
  - JoeQ is a lower-level IR than Java bytecode

- Operator_NullCheck
- Operator.BoundsCheck
- Operator.CheckCast, Operator.StoreCheck, etc.
JoeQ Operands \((joeq\text{-}Compiler\text{-}Quad\text{-}Operand)\)

- **RegisterOperand**
  - Abstract registers representing parameters, local variables, and temporal variables

- **ConstOperand**
  - int, float, long, double \textit{constants}

- **TargetOperand**
  - Basic block target of a branch instruction

- **MethodOperand, ParamListOperand**
  - Target and arguments to a method call

- **FieldOperand, TypeOperand, etc.**
JoeQ Quads

- Many useful methods, to get named registers, defined and used registers, etc
- See the assignment and JoeQ javadoc for more details
JoeQ CFGs \( (joeq.\text{Compiler}.\text{Quad}.\text{ControlFlowGraph}) \)

- Quads are organized into basic blocks
- Basic blocks are single-entry and also single-exit (barring exceptions)
- CFG, a control flow graph, is a graph of basic blocks with entry and exit
  - Entry and exit basic blocks always exist and are empty
  - Basic blocks are numbered with unique integer identifiers
  - Entry has identifier 0, exit has identifier 1
class ExprTest {
    int test (int a) {
        int b, c, d, e, f;
        c = a + 10;
        f = a + c;
        if (f > 2) {
            f = f - c;
        }
        return (f);
    }
    
    BB0 (ENTRY) (in: <none>, out: BB2)
    BB2 (in: BB0 (ENTRY), out: BB3, BB4)
    BB3 (in: BB2, out: BB4)
    BB4 (in: BB2, BB3, out: BB1 (EXIT))
    BB1 (EXIT) (in: BB4, out: <none>)
JoeQ Basic Blocks (`joeQ.Compiler.Quad.BasicBlock`)

- List of quads
- Provide access to successors and predecessors
- Exception control flow is not explicit in JoeQ basic blocks
  - An exception can jump out of the middle of a basic block
  - You do not need to consider exceptions in this class
Writing Analysis with JoeQ

Often can be written with *visitors*, that traverse all loaded CFGs, or all basic blocks or quads in those CFGs

- **ControlFlowGraphVisitor**: interface for an analysis that makes a pass over CFGs
- **BasicBlockVisitor**: interface for an analysis that makes a pass over all basic blocks
- **QuadVisitor**: interface for an analysis that makes a pass over quads
Visitor Design Pattern

- A clean interface to the complexities of JoeQ (Façade design pattern)
- Add an operation to existing objects without modifying the structure of objects
  - Operations: variable, Objects: fixed
- Control flow graph analysis 1, analysis 2, etc.
  - Add methods analysis1, analysis2, etc. to ControlFlowGraph class vs.
  - Visitor pattern: Analysis1CfgVisitor, Analysis2CfgVisitor, etc.
- In compiler, representation is (almost) fixed
- Adding analysis/transformation should be flexible
public class QuadCounter extends QuadVisitor.EmptyVisitor {
    public int count = 0;
    public void visitQuad(Quad q) { count++; }
}
QuadCounter

public class QuadCounter extends QuadVisitor. EmptyVisitor {
    public int count = 0;
    public void visitQuad(Quad q) { count++; }
}

- Some interfaces specify many methods
  - visitQuad will visit all quads
  - visitX will visit only specific quads
    - visitStore will visit only quads with Store operand
    - visitLoad will visit only quads with Load operand
- EmptyVisitor implements each method with a no-op
- Can override only the necessary methods
LoadStoreCounter

What if we only wanted to count the loads and stores?
public class QuadCounter extends QuadVisitor.EmptyVisitor {
    public int count = 0;
    public void visitQuad(Quad q) {
        count++;
    }
}

public class LoadStoreCounter extends QuadVisitor.EmptyVisitor {
    public int loadCount = 0, storeCount = 0;
    public void visitLoad(Quad q) { loadCount++; }
    public void visitStore(Quad q) { storeCount++; }
}
class CountQuads {
    public static void main(String[] args) {
        jq_Class[] classes = new jq_Class[args.length];
        for (String className : args) {
            jq_Class c = (jq_Class)Helper.load(className);
            System.out.println("Class:" + className);
            QuadCounter qc = new QuadCounter();
            Helper.runPass(c, qc);
            System.out.println(className + "has" + qc.count + "quads");
        }
    }
}

runPass(CFG or quad, visitor) runs a ControlFlowGraphVisitor or QuadVisitor over ControlFlowGraphs or Quads.
QuadIterator

- An alternative to visitors
- Simple interface to iterate through all the quads in a reverse post-order
- Extends `java.util.Iterator<Quad>`
QuadIterator

- An alternative to visitors
- Simple interface to iterate through all the quads in a reverse post-order
- Extends `java.util.Iterator<Quad>`

```java
ControlFlowGraph cfg = ...  
QuadIterator iter = new QuadIterator(cfg);
while (iter.hasNext()) {
    Quad quad = iter.next();
    if (quad.getOperator() instanceof Operator.Invoke) {
        doSomething(cfg.getMethod(), quad);
    }
}
```
QuadIterator

- next() and previous() return next and previous quad respectively in a reverse-post-order listing

- successors() and predecessors() return an iterator of possible successor quads and possible predecessor quads respectively
Java Beginners

- Java collections library
  - List, Set, Map, etc.
- Java generics
- Inner class
- ...
- Make yourself familiar with these concepts
HW 2: Dataflow framework

- We provide:
  - Solver interface: `Flow.Solver`
  - Analysis interface: `Flow.Analysis`
  - Two analyses that extend `Flow.Analysis`: `ConstantProp` and `Liveness`

- Goal is to complete:
  - MySolver skeleton that extends `Flow.Solver`
    - Should work with `ConstantProp` and `Liveness`
  - ReachingDefs skeleton that extends `Flow.Analysis`
  - Faintness analysis