Building a Compiler with JoeQ

AKA, how to solve Homework 2

Outline of this lecture

- Building a compiler: what pieces we need?
- An effective IR for Java
- joeQ
- Homework hints

How to Build a Compiler

1. Choose (or design) the language
2. Write the parser and AST structures
3. Write the frontend: symbol tables, type checking, IR generation
4. Middle end: machine independent optimizations in middle-end IR
    a. Loop optimizations
    b. Dataflow optimizations
    c. Peepholes
5. Back end: register allocation, block and instruction scheduling
6. Generate assembly code
7. Generate machine code and link with target runtime library

How to Build a Compiler

1. Choose (or design) the language
2. Write the parser and AST structures
3. Write the frontend: symbol tables, type checking, IR generation
4. Middle end: machine independent optimizations in middle-end IR
    a. Loop optimizations
    b. **Dataflow optimizations**
    c. Peepholes
5. Back end: register allocation, block and instruction scheduling
6. Generate assembly code
7. Generate machine code and link with target runtime library

How to Build a Compiler for Java

1. Reuse the JDK compiler (javac) for the parser and frontend
2. javac generates JVM bytecode
3. Should we use that for IR?

Pros and Cons of JVM Bytecode

- Pro: it’s machine independent
- Pro: it covers everything Java needs, and nothing more
- Pro: it’s high level, somewhat understandable, and has a lot of tooling

- Pro (for class purposes): javac does not perform any optimization

Cons: This is how JVM Bytecode Looks Like

```
int a, b, c;
    a = 3;
    if (a < 2) {
        b = 3;
        c = 5;
    } else {
        b = 3;
        c = 6;
    }
```

```
0:   iconst_3
1:   istore_1
2:   iload_1
3:   iconst_2
4:   if_icmpge    14
7:   iconst_3
8:   istore_2
9:   iconst_5
10:  istore_3
11:  goto    19
14:  iconst_3
15:  ...
```

Cons: JVM Bytecode is Stack-Oriented

```
int a, b, c;
  a = 3;
  if (a < 2) {
      b = 3;
      c = 5;
  } else {
      b = 3;
      c = 6;
  }
```

Bytecode	Meaning
0: iconst_3	push constant 3
1: istore_1	pop into reg 1
2: iload_1	push from reg 1
3: iconst_2	push constant 2
4: if_icmpge 14	pop 2 operands, cond jump
7: iconst_3	push constant 3
8: istore_2	pop into reg 2
9: iconst_5	push constant 5
10: istore_3	pop into reg 3
11: goto 19	uncond jump
14: iconst_3	push constant 3
15: …	…

Lots of pushes and pops

Moving code around is messy

Writing efficient JVM bytecode is hard

Also useless: HW uses registers not stack!

joeQ: a Register-Based IR for Java

```
int a, b, c;
    a = 3;
    if (a < 2) {
        b = 3;
        c = 5;
    } else {
        b = 3;
        c = 6;
    }
```

```
BB0 (ENTRY)    (in: <none>, out: BB2)
BB2    (in: BB0 (ENTRY), out: BB3, BB4)
1   MOVE_I                 R1 int,    IConst: 3
2   IFCMP_I                IConst: 3,    IConst: 2,
GE,    BB4
BB4    (in: BB2, out: BB5)
3   MOVE_I                 R2 int,    IConst: 3
4   MOVE_I                 R3 int,    IConst: 6
BB3    (in: BB2, out: BB5)
5   MOVE_I                 R2 int,    IConst: 3
6   MOVE_I                 R3 int,    IConst: 5
7   GOTO                   BB5
...
```

joeQ: a Register-Based IR for Java

- joeQ: joe Quads, but also “advanced” in Japanese (上級)
- *Quad*: a quadruple of (target, opcode, operand1, operand2):

ADD_I R1 int, R2 int, IConst: 3

Target: R1
Opcode: ADD_I (signed integer addition)
Operand 1: R2
Operand 2: immediate 3

- Also known as Three Address Code (chapter 6.2)

Properties of joeQ Quads

- Operands can be registers or constants
- **There are infinite registers!**

- All operands can be constants (no need for iconst-like operations)
  - Simplifies constant folding
  - Makes everything else harder

What joeQ Quads Are There?

- Moves
- Arithmetic and conversion ops (UnaryOperator, BinaryOperator)
- Memory ops (GETFIELD, PUTFIELD, ALOAD, ASTORE)
- Function calls (INVOKEVIRTUAL, INVOKESPECIAL, INVOKEINTERFACE)
- Control flow (cond jumps, table switches)
- Special Java operations (MONITORENTRY/EXIT, INSTANCEOF, etc.)
- A few weird ones (PEEK, POKE)

Exception handling in joeQ

- joeQ quads never fail
- Except for those who are specified to fail:
  - NULL_CHECK (throws NullPointerException)
  - ZERO_CHECK (throws DivisionByZeroException)
  - BOUNDS_CHECK (throws ArrayIndexOutOfBoundsException)
  - ASTORE_CHECK (throws ArrayStoreException)

- Exceptions, even from the CHECK quads, are never modeled in CFG
- Don’t need to worry in this class

What joeQ Quads Are There, Really?

- The Full List:
  - http://joeq.sourceforge.net/apidocs/joeq/Compiler/Quad/Operator.html

- To find out what an Operator does:
  - Check if the name matches a JVM bytecode (https://docs.oracle.com/javase/specs/jvms/se7/html/index.html)
  - Read the implementation Operator.interpret()
  - Ask in Piazza or at OH

Manipulating Quads

- The Quad class:
  http://joeq.sourceforge.net/apidocs/joeq/Compiler/Quad/Quad.html
- getOperator() : find what the quad does
- getOperands() : find inputs to the quads
- getDefinedRegisters() : find where the registers written by the quad

- NOTE: despite the name, a quad can have more than 2 operands!

Manipulating Quads

- Quads are arranged in a ControlFlowGraph
  - http://joeq.sourceforge.net/apidocs/joeq/Compiler/Quad/ControlFlowGraph.html
- Composed of BasicBlock
  - http://joeq.sourceforge.net/apidocs/joeq/Compiler/Quad/BasicBlock.html
- The joeQ loader constructs the CFG for you
- Iterate quads in BB with a QuadIterator or with a QuadVisitor
  - http://joeq.sourceforge.net/apidocs/joeq/Compiler/Quad/QuadIterator.html
  - http://joeq.sourceforge.net/apidocs/joeq/Compiler/Quad/QuadVisitor.html

Object Oriented Compiler Design

- One Object for each IR element
  - Quads, BasicBlock,
- One Object for each compiler phase
  - A Parser, a Frontend, an Optimizer, a Backend, etc.
- One Object for each analysis (eg. DeadCodeElimination, or EnterSSA)
  - Analyze and transform IR, often with visitor pattern

Object Oriented Compiler Design

- One Object for each IR element
  - Quads, BasicBlock,
- One Object for each compiler phase
  - A Parser, a Frontend, an Optimizer, a Backend, etc.
  - **Your goal in HW: write a generic dataflow analyzer, with many pluggable analyses**
  - Should work with the example analysis (Liveness, Constant Propagation)
- One Object for each analysis (eg. DeadCodeElimination, or EnterSSA)
  - Analyze and transform IR, often with visitor pattern
  - **Your goal in HW: write dataflow analyses for Faintness and Reaching Definitions**

Homework Details

- We provide a skeleton, testing code and two example analyses

- The handout details how to set up joeQ
- We’ll go through that in more details on Friday

What parts of joeQ can you use?

- Almost all of joeQ is allowed
- Most of it won't be useful though (eg the bytecode loader, the interpreter, or the runtime emulator)

- **Do not use joeq.Compiler.Dataflow**
- For this homework, also do not use joeq.Compiler.Quad.SSA
- No external libraries, and no Java > 1.5 APIs or syntax
- Check on myth with Java 1.5 that your code works

Why joeQ?

- joeQ is not a learning goal for this class
  - Unlike say, Java in 106A, Haskell in 242/240H or Tensorflow in 224N/231N
- So why not learn LLVM instead?

- The same concepts apply to any mid-level IR (plus or minus SSA)
- LLVM is vast and complicated, because C++ is vast and complicated