

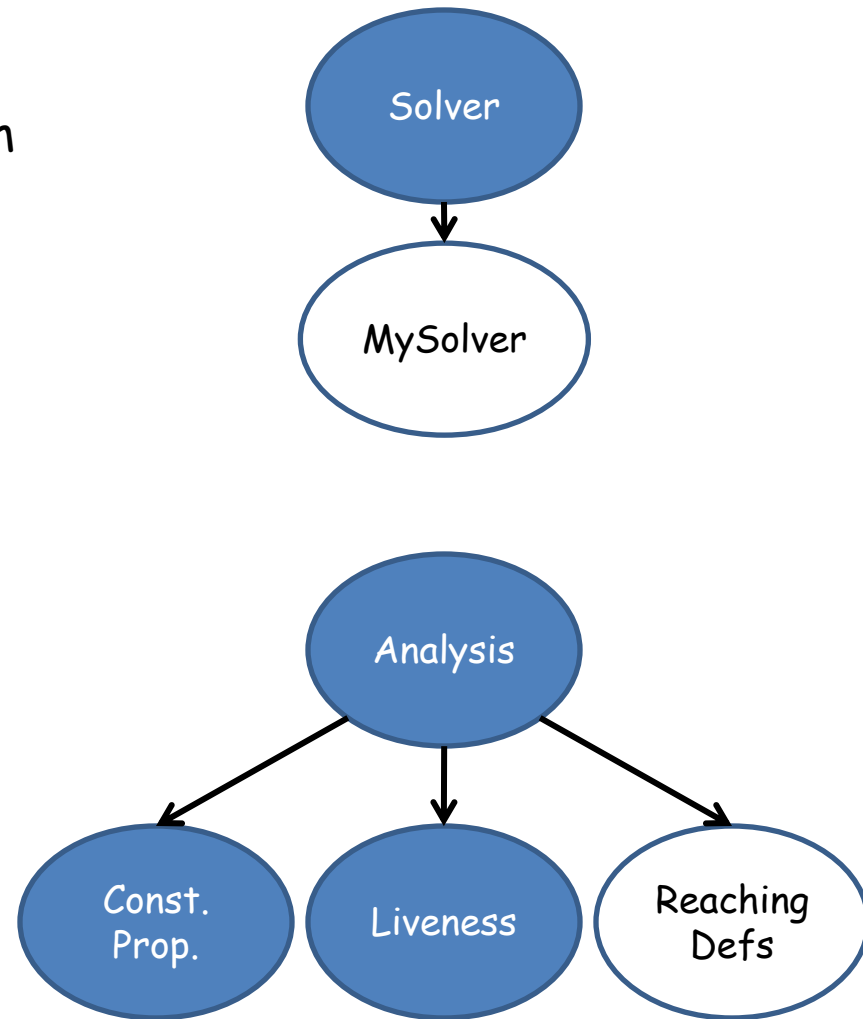


Joeq Framework

CS 243, Winter 2010-2011

Homework 2: Dataflow Framework

- Solver interface
 - Represent iterative dataflow algorithm
 - Used for multiple dataflow analyses
- Analysis interface
 - Specify per-analysis properties: direction, lattice value, boundary condition, ...
 - e.g., Constant Propagation, Liveness, Reaching Definition



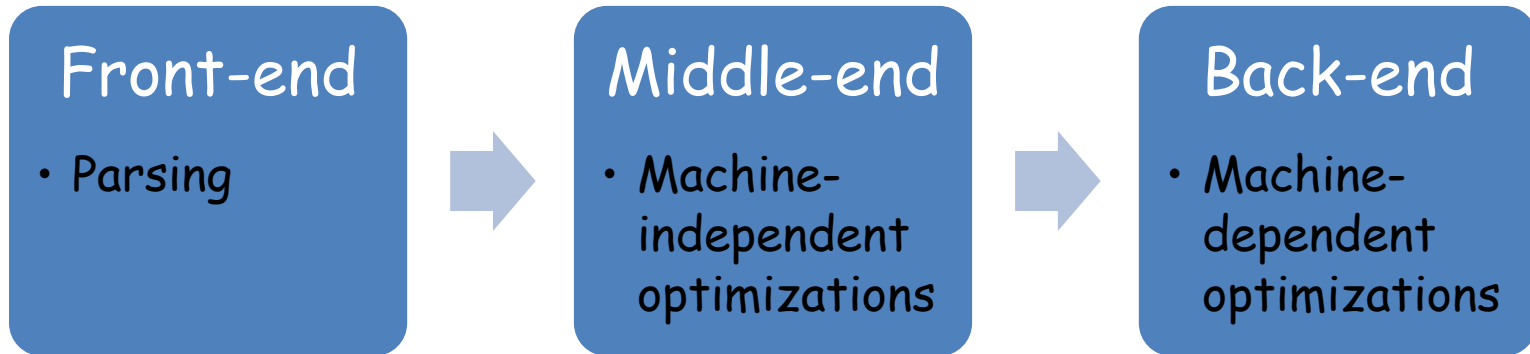
Joeq

- 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”
 - *Advanced* level in Japanese (上級)

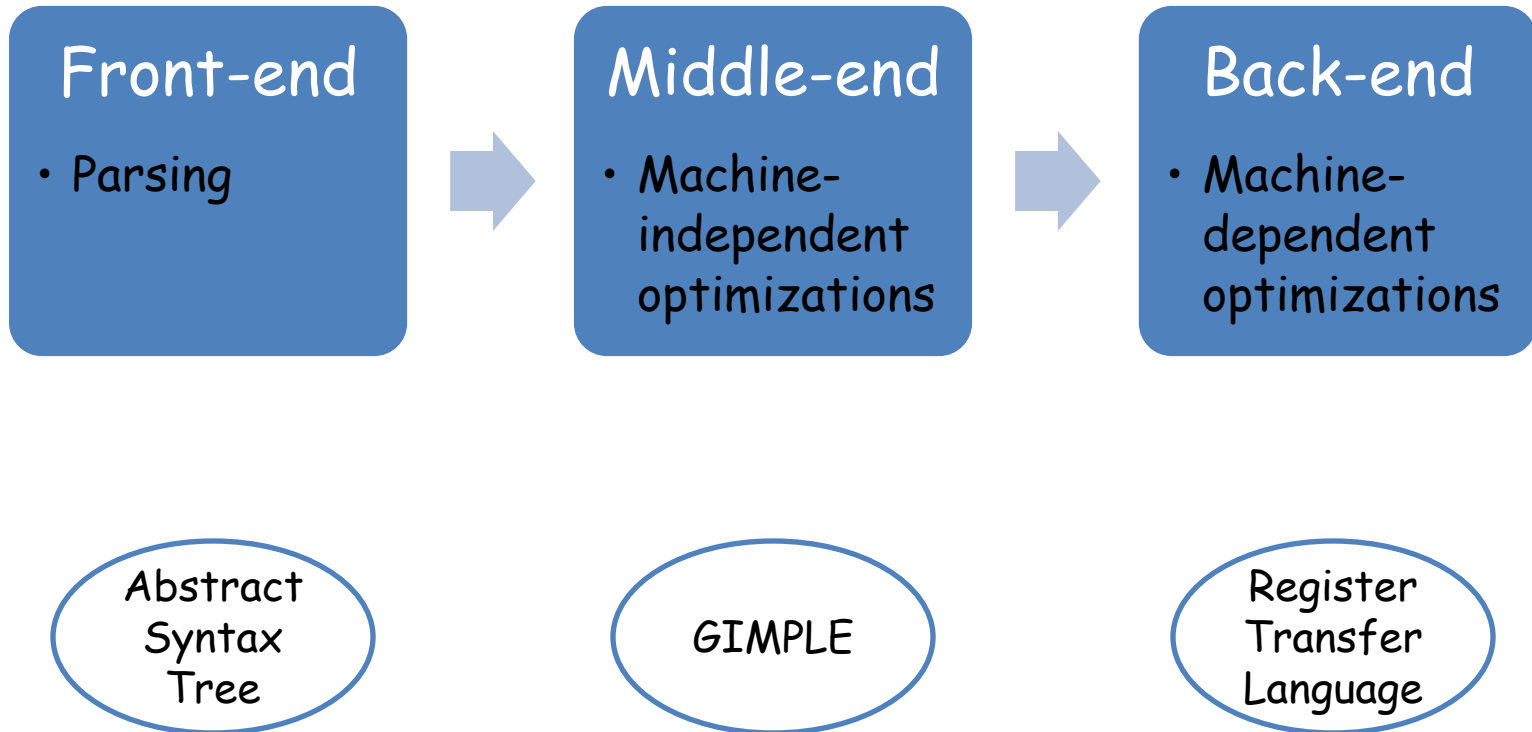
Overview

- Intermediate representation in real compilers
- Joeq framework focusing on its intermediate representation
 - Java bytecode
 - Joeq quads: instruction set used in Joeq
- Homework 2

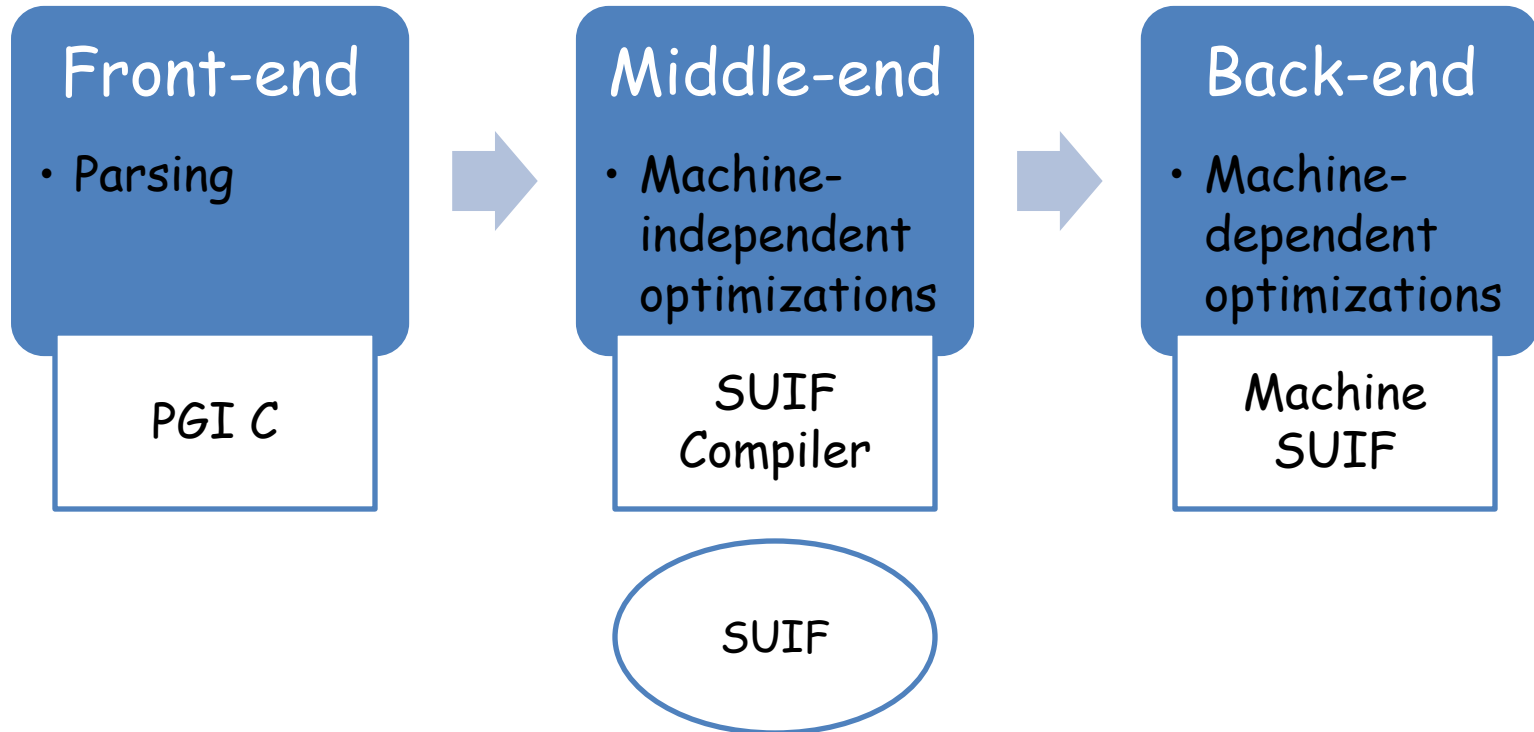
Typical Compiler Infrastructure



GCC

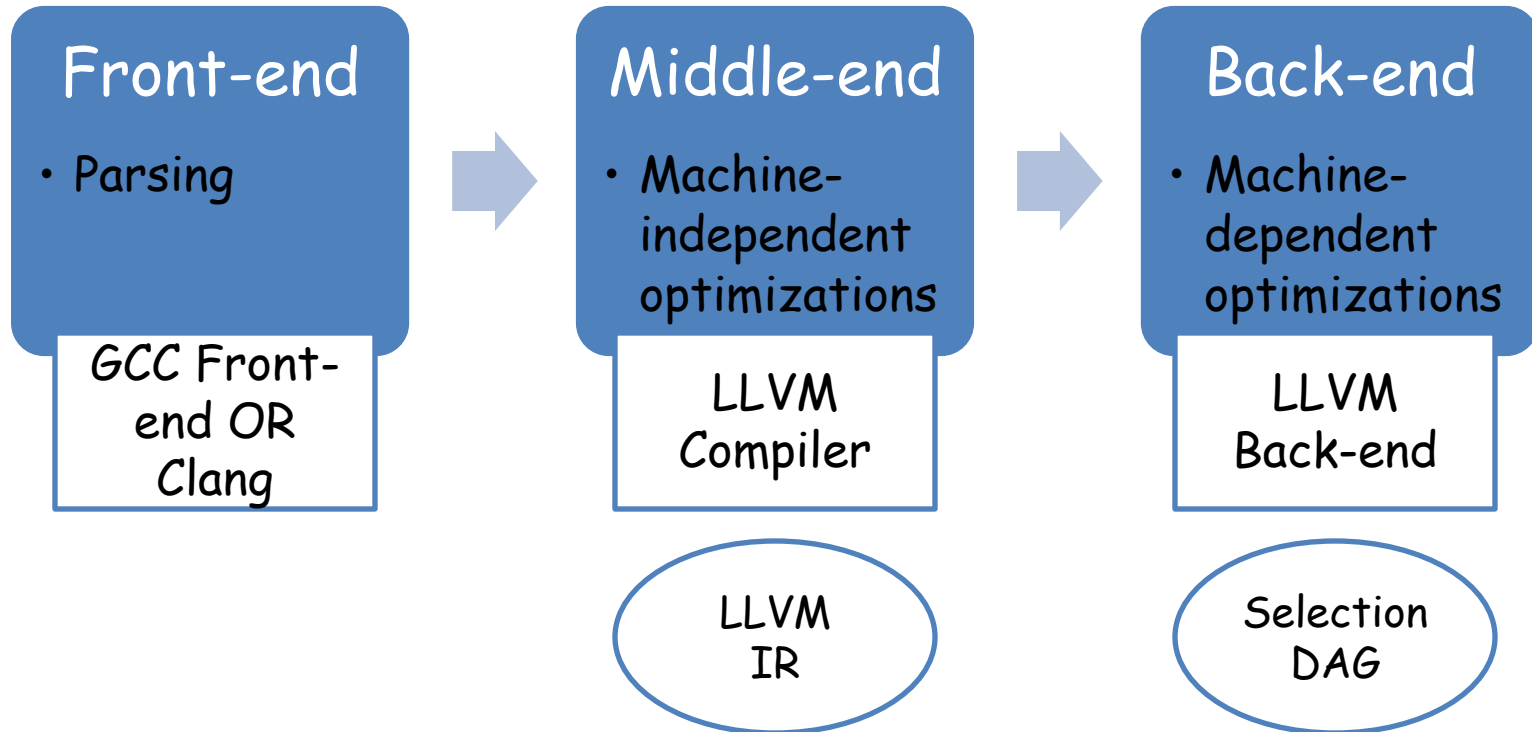


SUIF: Stanford University Intermediate Format



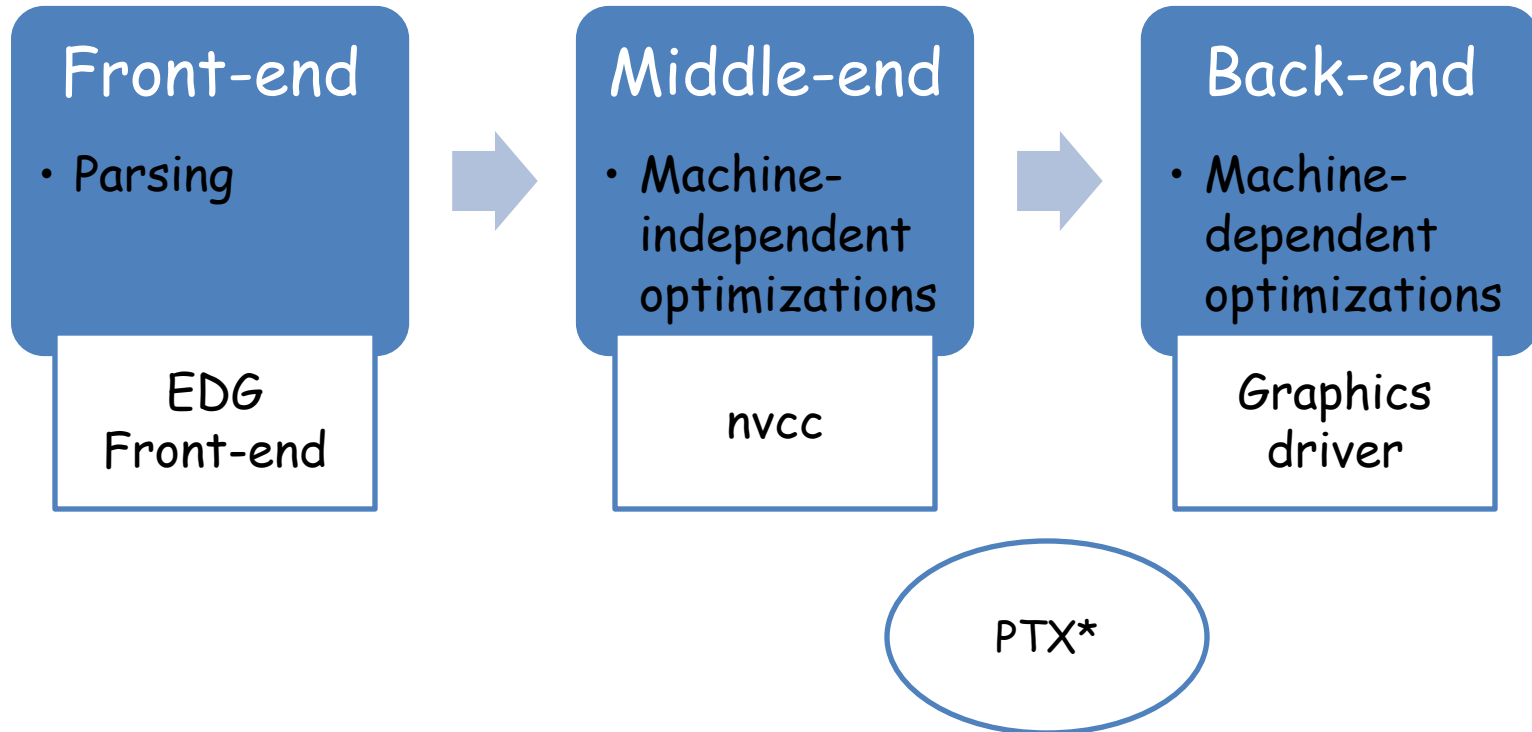
- Persistent and consistent intermediate representation

LLVM: Low-Level Virtual Machine (UIUC)



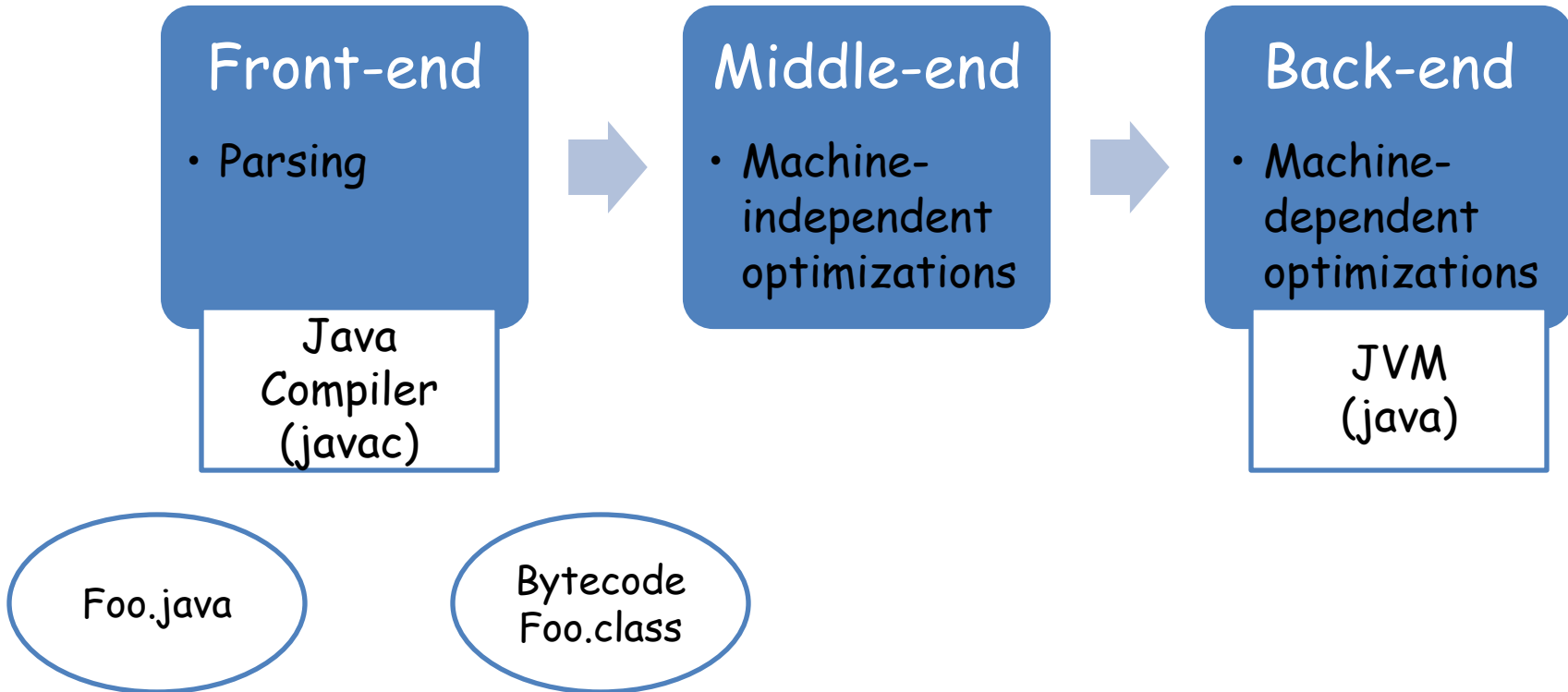
- Apple and NVIDIA OpenCL* compilers
- KLEE: symbolic execution for automatic test generation
- *OpenCL: A framework for programming heterogeneous platforms

nvcc: NVIDIA CUDA Compiler

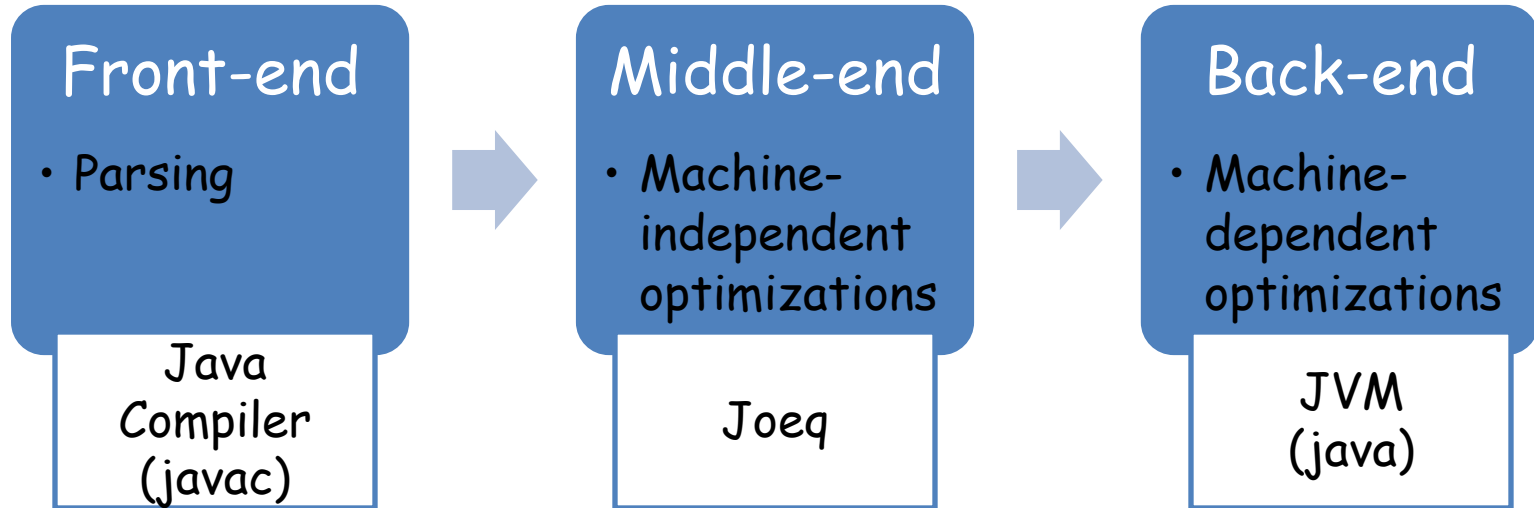


* PTX: Parallel Thread Execution

Java



Joeq



Foo.java

Bytecode
Foo.class

Quad

Java Source Code

- Input to the Java Front-end
- A very “rich” representation
 - Good for reading and writing (by human)
 - Hard to analyze (by computer)
- Many high-level concepts with no hardware counterparts
 - classes, generics, virtual function calls, exceptions, structured control flow, locks, etc.

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
- Still high level
 - Virtual methods, locks, etc.

Java Bytecode Representation

- Each bytecode instruction uses one byte
 - Instructions may have additional operands, stored immediately after the instruction
- Variables stored in abstract registers
 - r0 is this
 - r1, ... are parameters followed by locals
- Stack machine model
 - All intermediate values stored on a stack

Stack Machine Model

- Instructions push/pop values onto a stack
- $x = y + 10 \rightarrow$
 - push y
 - push 10
 - add
 - pop 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
- `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

Example: ExprTest

```
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():
```

```
    Code:
```

```
    ...
```

```
  int test(int):
```

```
    Code:
```

```
    ...
```

Example: ExprTest.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

Example: 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;
  }
}
```

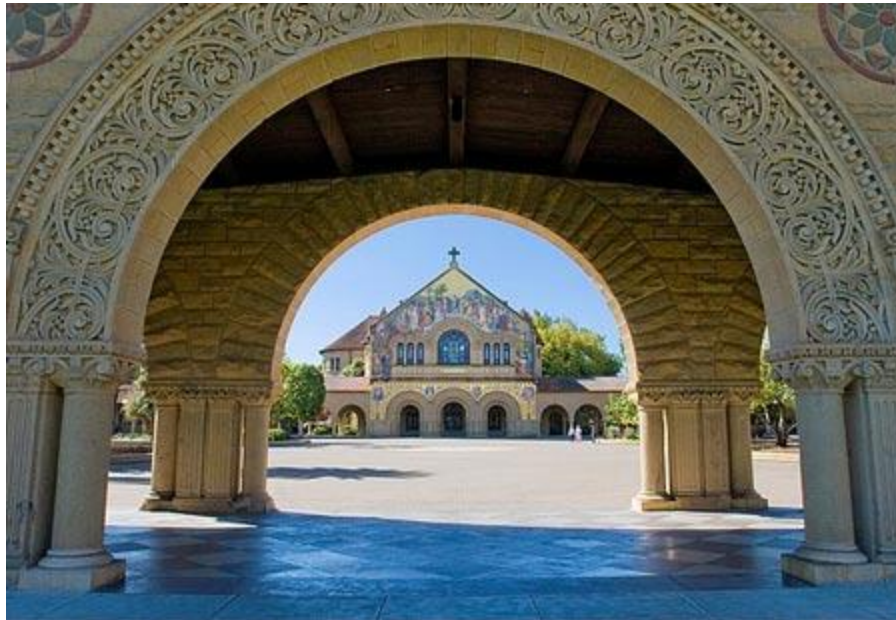
```
int test(int)
Code:
0:   iload_1
1:   bipush  10
3:   iadd
4:   istore_3
5:   iload_1
6:   iload_3
7:   iadd
8:   istore  6
10:  iload   6
12:  iconst_2
13:  if_icmple 22
16:  iload   6
18:  iload_3
19:  isub
20:  istore  6
22:  iload   6
24:  ireturn
```

Joeq Intermediate Representation

- High-level representation: Joeq has classes for each component of the Java .class file
 - `jq_Type`, `jq_Class`, `jq_Field`, `jq_Method`, ...
- We use a lower-level representation called "Quad".
- Joeq translates bytecodes into quads.

Quads: i.e., Stanford Version of Three Address Code (TAC)

- One operator and up to four operands: **four**-address instructions
- `joeq.Compiler.Quad.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 lower-level IR than Java Bytecode



Joeq Operands (joeq.Compiler.Quad.Operand)

- Register Operand
 - Abstract registers representing parameters, local variables, and temporal variables
- Constant Operand
 - int/float/string/etc. constants
- Target Operand
 - Basic block target of a branch instruction
- Method Operand, ParamListOperand
 - Target and arguments to a method call
- Field Operand, TypeOperand, ...

Joeq Operators (joeq.Compiler.Quad.Operator)

- Operator.Move
- Operator.Unary, Operator.Binary
- Operator.Invoke
- Operator.Branch, Operator.GetField, Operator.PutField, Operator.New, ...
- Have suffixes indicating return type
 - ADD_I adds two integers
 - L, F, D, A, and V refer to long, float, double, reference, and void

Joeq Runtime Checking Operators

- Runtime checks are explicit quads
 - Not implicit as in bytecodes: Joeq is lower-level IR than Java Bytecode
 - Related to the second programming assignment (HW4)
- `Operator.NullCheck`
- `Operator.BoundsCheck`
- `Operator.CheckCast`, `Operator.StoreCheck`, ...

Joeq CFGs (joeq.Compiler.Quad.ControlFlowGraph)

- Graphs of basic blocks with entry and exit
 - Entry and exit basic blocks always exist.
 - They are empty.

Joeq Basic Blocks (joeq.Compiler.Quad.BasicBlock)

- Lists 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

Example: ExprTest

```
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
> java PrintQuads ExprTest
```

Class: ExprTest

Control flow graph for
ExprTest.<init> ()V:

...

Control flow graph for
ExprTest.test (I)I:

...

Example: 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
```

Bytecode

BB0 (ENTRY) (in: <none>, out: BB2)

BB2 (in: BB0 (ENTRY), out: BB1 (EXIT))

```
1  NULL_CHECK      T-1 <g>,                R0 ExprTest
2  INVOKESPECIAL_V% java.lang.Object.<init>()V, (R0 ExprTest)
3  RETURN_V
```

Quads

BB1 (EXIT) (in: BB2, out: <none>)

Example: ExprTest.test(int a)

BB0 (ENTRY) (in: <none>, out: BB2)

```
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;
  }
}
```

BB2 (in: BB0 (ENTRY), out: BB3, BB4)

```
1  ADD_I   T2 int, R1 int,   IConst: 10
2  MOVE_I  R3 int, T2 int
3  ADD_I   T2 int, R1 int,   R3 int
4  MOVE_I  R4 int, T2 int
5  IFCMP_I R4 int, IConst: 2, LE,          BB4
```

BB3 (in: BB2, out: BB4)

```
6  SUB_I   T2 int, R4 int, R3 int
7  MOVE_I  R4 int, T2 int
```

BB4 (in: BB2, BB3, out: BB1 (EXIT))

```
8  RETURN_I R4 int
```

BB1 (EXIT) (in: BB4, out: <none>)

Example: ExprTest.test(int a)

```
0:  iload_1          BB0 (ENTRY) (in: <none>, out: BB2)
1:  bipush  10
3:  iadd             BB2      (in: BB0 (ENTRY), out: BB3, BB4)
4:  istore_3         1  ADD_I    T2 int, R1 int,   IConst: 10
5:  iload_1          2  MOVE_I    R3 int, T2 int
6:  iload_3          3  ADD_I    T2 int, R1 int,   R3 int
7:  iadd             4  MOVE_I    R4 int, T2 int
8:  istore  6        5  IFCMP_I  R4 int, IConst: 2, LE,          BB4
10: iload  6
12: iconst_2        BB3      (in: BB2, out: BB4)
13: if_icmple 22    6  SUB_I    T2 int, R4 int, R3 int
16: iload  6        7  MOVE_I    R4 int, T2 int
18: iload_3
19: isub            BB4      (in: BB2, BB3, out: BB1 (EXIT))
20: istore  6        8  RETURN_I R4 int
22: iload  6
24: ireturn        BB1 (EXIT) (in: BB4, out: <none>)
```

Writing Analysis with Joeq

- Often can be written with visitors
 - Traverse all the loaded CFGs, or all the quads in those CFGs
- `ControlFlowGraphVisitor`: interface for an analysis which makes a pass over the CFGs
- `QuadVisitor`: interface for an analysis which makes a single pass over the quads in a CFG

Analysis Example: QuadCounter and LoadStoreCounter

```
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++; }
}
```

Visitor Design Pattern

- Add an operation to existing objects without modifying the structure of objects
- Operations: variable, objects: fixed
- Control flow graph analysis 1, 2, ...
 - Add methods `analysis1`, `analysis2`, ... to `ControlFlowGraph` class
 - vs.
 - `Analysis1CfgVisitor`, `Analysis2CfgVisitor`, ...
- In compiler, representation is (almost) fixed.
- Adding analysis/transformation should be flexible.

joeq.Main.Helper class

- A clean interface to the complexities of Joeq (Façade design pattern)
- `runPass(CFG or quad, visitor)` runs a `ControlFlowGraphVisitor/QuadVisitor` over `ControlFlowGraphs/Quads`.

Helper Class Usage Example: QuadCounter

```
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");
        }
    }
}
```

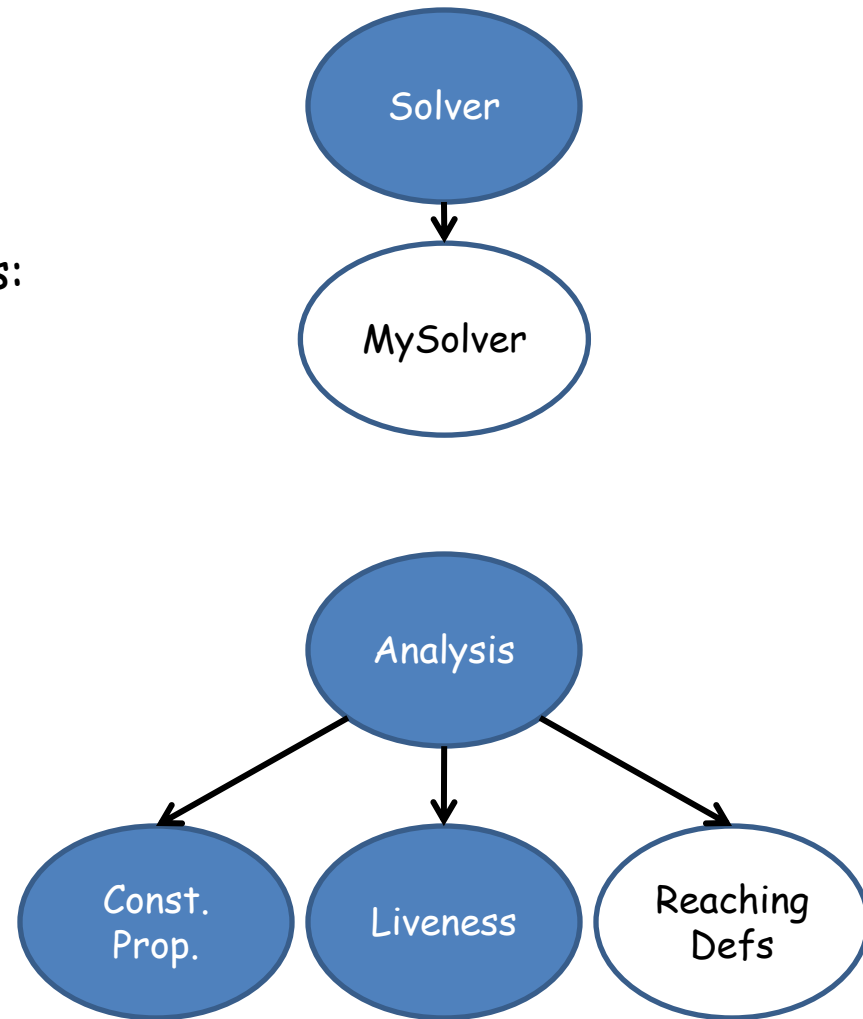
QuadIterator

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

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

Homework 2: Dataflow Framework

- We provide
 - Solver interface: `Flow.Solver`
 - Analysis interface: `Flow.Analysis`
 - Two analysis that extend `Flow.Analysis`: `ConstantProp` and `Liveness`
- Goal is to complete
 - Skeleton `MySolver` that extends `Flow.Solver`
 - Should work with `ConstantProp` and `Liveness`
 - Skeleton `ReachingDefs` that extends `Flow.Analysis`



For Java Beginners

- Java collections library
 - List, Set, Map, ...
- Java Generics
- Inner Class
- ...
- Make yourself familiar with these concepts.

Homework 2: Details

- Due: Next Friday (1/28), 5PM, PST
- joeq.jar is provided
 - Unjar this if you want to look at Joeq source code.
- Using Eclipse may make your life easier.
- Working with groups of two is encouraged.
- Output must match ours on Stanford Linux clusters such as myth.

- **Get started early.** It make take a long time to understand the Joeq framework, although you need to know only small part of it.
- Post questions on Piazza, so that we can answer them the session on Friday.