Sample “Optimizations”

**Constant propagation and folding**

```java
a = 1;
b = 2;
c = a + b;
c = 3;
```

**Common subexpression elimination**

```java
a = b + c;
d = b + c;
a = b + c;
d = a;
```

**Unreachable code elimination**

```java
if(DEBUG)
System.out.println("\n");
```
Sample “Optimizations”

Arithmetic optimizations

\[
x = y \times 1; \\
i = j + j; \\
a = b\times c + b\times d;
\]

\[
x = y; \\
i = j \ll 1; \\
a = b\times (c + d);
\]

Loop-invariant code motion

```java
for(i = 0; i < a.length - foo; i++) {
    sum += a[i];
}
```

```java
l = a.length - foo;
for(i = 0; i < l; i++) {
    sum += a[i];
}
```
Check elimination

```java
for(i = 0; i < 10; i++) {
    if(a == null) throw new Exception();
    if(i<0 || i>=a.length) throw new Exception();
    a[i] = i;
}
```

```java
for(i = 0; i < 10; i++) {
    a[i] = i;
}
```
Full Employment Theorem for Compiler Writers

No compiler produces optimal correct code for all programs. For every compiler \( A \), there exists a better compiler \( B \).

So, “optimizing” compiler really means (hopefully) improving compiler.
“Optimizations” can hurt performance

```plaintext
a = b + c;
d = b + c;
a = b + c;
d = a;
```
“Optimizations” can hurt performance

```c
a = b + c;  // expensive computation requiring many registers
d = b + c;
```

```c
a = b + c;  // expensive computation requiring many registers
d = a;
```
Hardware performance can be hard to model

Gu, Verbrugge, Gagnon, Code Layout as a Source of Noise in JVM Performance, Component And Middleware Performance Workshop, OOPSLA 2004
An “optimization” should be...

- semantics-preserving (“safe”)
- profitable
- widely applicable
- cheap to perform
  - compilation time
  - memory requirements
  - implementation complexity
while (i < 10) {
    sum = sum + 2 * a[i];
}
while(i<10) { sum = sum + 2*a[i]; }

Three-address code sequence:

L0:
t1 = i >= 10;
if t1 goto L1;
t2 = i * 4;
t3 = a + t2;
t4 = *t3;
t5 = 2 * t4;
sum = sum + t5;
goto L0;
L1:
while (i < 10) {
    sum = sum + 2 * a[i];
}
### Intermediate representations

<table>
<thead>
<tr>
<th>High-level</th>
<th>↔</th>
<th>Low-level</th>
</tr>
</thead>
<tbody>
<tr>
<td>language-specific</td>
<td>↔</td>
<td>language-independent</td>
</tr>
<tr>
<td>machine-independent</td>
<td>↔</td>
<td>machine-specific</td>
</tr>
<tr>
<td>tree/graph</td>
<td>↔</td>
<td>instruction sequence</td>
</tr>
<tr>
<td>structured control flow</td>
<td>↔</td>
<td>gotos</td>
</tr>
<tr>
<td>compound expressions</td>
<td>↔</td>
<td>simple expressions</td>
</tr>
<tr>
<td>high-level constructs</td>
<td>↔</td>
<td>constructs expanded</td>
</tr>
</tbody>
</table>
Intermediate representations

source

HIR

MIR

LIR

target
Q1: What is the output of this program?

System.out.println("Hello, World!");
Q1: What is the output of this program?

System.out.println("Hello, World!");

Q2: Given an arbitrary program $p$, can you tell whether its output is “Hello, World!”?
if( arbitraryComputation() ) {
    System.out.println("Hello, World!");
} else {
    System.out.println("Goodbye");
}
Does this program cause an array overflow?

if( arbitraryComputation() ) {
    int a[] = new int[5];
    a[10] = 10;
}
For any interesting property $Pr$ of the behaviour of a program, it is impossible to write an analysis that can decide for every program $p$ whether $Pr$ holds for $p$. 
Static Analysis

We settle for static analyses that approximate a property $Pr$. Example: Does program $p$ access an array out of bounds?

It’s always safe to say “maybe”!
Static Analysis

Sound
An analysis is sound if its result includes every possible behaviour (but may include additional behaviours).

Conservative
An analysis is conservative if its result includes every possible behaviour (but may include additional behaviours).
i = 1
L3: if i > 1 goto L1
    a = 2
    b = 3
    goto L2
L1: a = 1
    b = 4
L2: c = a + b
    i = i + 1
    goto L3
Control Flow Graph

i = 1

L3: if i > 1 goto L1

F  T

a = 2

L1: a = 1

b = 3

b = 4

goto L2

L2: c = a + b

i = i + 1

goto L3
Basic Block Graph

L3: if i > 1 goto L1

F

T

a = 2
b = 3
goto L2

L1: a = 1
b = 4

L2: c = a + b
i = i + 1
goto L3