Introduction to Code Generation
Instructor: Hossein Hojjat
April 7, 2017
Compiler Phases

Source Code (concrete syntax)
if (x == 0) x = x + 1;

Token Stream
if ( x == 0 ) x = x + 1;

Abstract Syntax Tree (AST)

Attributed AST

Machine Code
16: iload_2
17: ifne 24
20: iload_2
21: iconst_1
22: iadd
23: istore_2
24: ...

Lexical Analysis
Syntax Analysis (Parsing)
Semantic Analysis (Name Analysis, Type Analysis, ...)
Code Generation
Error
Code Generation Example

- Phase after type checking emits such bytecode instructions

```java
while (i > 0) {
  i += 2 * j + 1;
  j = j - 5;
  System.out.println(j);
}
```

```
5: iload_1
6: ifle 31
9: iload_1
10: iconst_2
11: iload_2
12: imul
13: iconst_1
14: iadd
15: iadd
16: istore_1
17: iload_2
18: iconst_5
19: isub
20: istore_2
21: getstatic #2 // System.out
24: iload_2
25: invokevirtual #3 // println
28: goto 5
31: // ...
```
Java Virtual Machine (JVM)

- Programs are written in Java or other languages
- Compiler translates them to Java Bytecode
- Platform-specific Java Virtual Machine executes Java Bytecode

A.java  B.py  C.scala
javac  jython  scalac
A.class  B.class  C.class
Java Bytecode
(Platform Independent)

Platform-Specific Java Virtual Machine (JVM)
Java Virtual Machine (JVM)

- JVM is a stack machine: evaluation of expressions uses a stack (operand stack)
- Instructions fetch their arguments from the top of the operand stack
- Instructions store their results at the top of the operand stack
Why a Stack Machine

- A simple evaluation model: no variables or registers
- Each operation:
  - takes operands from top of stack
  - puts results back at top of stack
- Instruction “add” as opposed to “add r1, r2”
- Simpler compiler, more compact programs
Java Statement: \( i \ += 2 \times j + 1; \)
Java Statement: $i += 2 \times j + 1$;

Java Bytecode:

6: // ...
9: iload_1
10: iconst_2
11: iload_2
12: imul
13: iconst_1
14: iadd
15: iadd
16: istore_1
17: // ...
Java Bytecode
6: // ...
9: iload_1
10: iconst_2
11: iload_2
12: imul
13: iconst_1
14: iadd
15: iadd
16: istore_1
17: // ...

Java Statement: i += 2 * j + 1;
Java Statement: \( i += 2 \times j + 1; \)
Java Statement: \( i += 2 \times j + 1; \)
Java Statement: i += 2 * j + 1;
Java Bytecode

6: // ...
9: iload_1
10: iconst_2
11: iload_2
12: imul
13: iconst_1
14: iadd
15: iadd
16: istore_1
17: // ...

Java Statement: i += 2 * j + 1;
Java Bytecode

6: // ...
9: iload_1
10: iconst_2
11: iload_2
12: imul
13: iconst_1
14: iadd
15: iadd
16: istore_1
17: // ...

PC

memory:

SP

Java Statement: i += 2 * j + 1;
Java Statement: $i += 2 \times j + 1$;
Instructions in JVM

- Separate for each type, including
  - integer types (iadd, imul, iload, istore, bipush)
  - reference types (aload, astore)

- Why are they separate if not in e.g. x86?
  - Memory safety
  - Each reference points to a valid allocated object

- Conditionals and jumps

- Further high-level operations
  - array operations
  - object method and field access
http://docs.oracle.com/javase/specs/jvms/se8/html/index.html

- Use javac -g *\.java to compile
- Use javap -c -l ClassName to explore
<table>
<thead>
<tr>
<th>Instruction</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>iload_x</td>
<td>Loads the integer value of the local variable in slot $x$ on the stack.</td>
</tr>
<tr>
<td></td>
<td>$x \in {0, 1, 2, 3}$</td>
</tr>
<tr>
<td>iload X</td>
<td>Loads the value of the local variable pointed to by index $X$ on the top of</td>
</tr>
<tr>
<td></td>
<td>the stack.</td>
</tr>
<tr>
<td>iconst_x</td>
<td>Loads the integer constant $x$ on the stack. $x \in {0, 1, 2, 3, 4, 5}$</td>
</tr>
<tr>
<td>bipush X</td>
<td>Like iconst, but for arbitrarily large $X$.</td>
</tr>
<tr>
<td>istore_x</td>
<td>Stores the current value on top of the stack in the local variable in slot</td>
</tr>
<tr>
<td></td>
<td>$x \in {0, 1, 2, 3}$</td>
</tr>
<tr>
<td>istore X</td>
<td>Stores the current value on top of the stack in the local variable indexed</td>
</tr>
<tr>
<td></td>
<td>by $X$.</td>
</tr>
<tr>
<td>ireturn</td>
<td>Method return statement (note that the return value has to have been put</td>
</tr>
<tr>
<td></td>
<td>on the top of the stack beforehand).</td>
</tr>
<tr>
<td>iadd</td>
<td>Pop two (integer) values from the stack, add them and put the result back</td>
</tr>
<tr>
<td></td>
<td>on the stack.</td>
</tr>
<tr>
<td>isub</td>
<td>Pop two (integer) values from the stack, subtract them and put the result</td>
</tr>
<tr>
<td></td>
<td>back on the stack.</td>
</tr>
</tbody>
</table>
## Selected Instructions

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>imult</strong></td>
<td>Pop two (integer) values from the stack, multiply them and put the result back on the stack.</td>
</tr>
<tr>
<td><strong>idiv</strong></td>
<td>Pop two (integer) values from the stack, divide them and put the result back on the stack.</td>
</tr>
<tr>
<td><strong>irem</strong></td>
<td>Pop two (integer) values from the stack, put the result of $x_1 % x_2$ back on the stack.</td>
</tr>
<tr>
<td><strong>ineg</strong></td>
<td>Negate the value on the stack.</td>
</tr>
<tr>
<td><strong>iinc x, y</strong></td>
<td>Increment the variable in slot $x$ by amount $y$.</td>
</tr>
<tr>
<td><strong>ior</strong></td>
<td>Bitwise OR for the two integer values on the stack.</td>
</tr>
<tr>
<td><strong>iand</strong></td>
<td>Bitwise AND for the two integer values on the stack.</td>
</tr>
<tr>
<td><strong>ixor</strong></td>
<td>Bitwise XOR for the two integer values on the stack.</td>
</tr>
</tbody>
</table>
| **ifXX L**  | Pop one value from the stack, compare it zero according to the operator $XX$. If the condition is satisfied, jump to the instruction given by label $L$.  
$XX \in \{eq, lt, le, ne, gt, ge, null, nonnull\}$ |
## Selected Instructions

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>if_icmpXX L</code></td>
<td>Pop two values from the stack and compare against each other. Rest as <code>ifXX L</code>.</td>
</tr>
<tr>
<td><code>goto L</code></td>
<td>Unconditional jump to instruction given by the label <code>L</code>.</td>
</tr>
<tr>
<td><code>pop</code></td>
<td>Discard word currently on top of the stack.</td>
</tr>
<tr>
<td><code>dup</code></td>
<td>Duplicate word currently on top of the stack.</td>
</tr>
<tr>
<td><code>swap</code></td>
<td>Swaps the two top values on the stack.</td>
</tr>
<tr>
<td><code>aload_x</code></td>
<td>Loads an object reference from slot <code>x</code>.</td>
</tr>
<tr>
<td><code>aload X</code></td>
<td>Loads an object reference from local variable indexed by <code>X</code>.</td>
</tr>
<tr>
<td><code>iaload</code></td>
<td>Loads onto the stack an integer from an array. The stack must contain the array reference and the index.</td>
</tr>
<tr>
<td><code>iastore</code></td>
<td>Stores an integer in an array. The stack must contain the array reference, the index and the value, in that order.</td>
</tr>
</tbody>
</table>
Example: Twice

class Expr1 {
    public static int twice(int x) {
        return x*2;
    }
}

> javac -g Expr1.java; javap -c -l Expr1

public static int twice(int);

Code:
0: iload_0    // load int from var 0 to top of stack
1: iconst_2   // push 2 on top of stack
2: imul       // replace two topmost elements with their product
3: ireturn    // return top of stack
Example: Area

class Expr2 {
    public static int cubeArea(int a, int b, int c) {
        return (a*b + b*c + a*c) * 2;
    }
}

> javac -g Expr2.java; javap -c -l Expr2

LocalVariableTable:

<table>
<thead>
<tr>
<th>Start</th>
<th>Length</th>
<th>Slot</th>
<th>Name</th>
<th>Signature</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>14</td>
<td>0</td>
<td>a</td>
<td>I</td>
</tr>
<tr>
<td>0</td>
<td>14</td>
<td>1</td>
<td>b</td>
<td>I</td>
</tr>
<tr>
<td>0</td>
<td>14</td>
<td>2</td>
<td>c</td>
<td>I</td>
</tr>
</tbody>
</table>

public static int cubeArea(int, int, int);
Code:
0: iload_0
1: iload_1
2: imul
3: iload_1
4: iload_2
5: imul
6: iadd
7: iload_0
8: iload_2
9: imul
10: iadd
11: iconst_2
12: imul
13: ireturn