Compilers vs. Interpreters

- Compiler: Translate once, then run the result e.g. translate C++ into 80x86 machine language
  → stand-alone code, faster execution

- Interpreter: Read-check-execute loop, e.g. CAML or Prolog environment
  → good for learning/debugging, (slower, not stand-alone)
Intermediate Representation

• Like an abstract assembly language, for an imaginary machine

• Make no assumptions about target architecture (e.g. number of registers, what operations are fastest etc.)

• But ... make it “look like” the source language
Three (old) reasons to use an IR

1. Re-targeting a compiler onto a different architecture

2. Speeding up the loading of library files in an interpreter

3. “Half-way” between source and target language - make the compiler/interpreter's job a bit easier
1. Re-targeting a Compiler

- Have e.g. a C to 80x86 compiler

- Want a C to Sparc compiler
  → Should only have to re-write the back end

- Want a C++ to 80x86 compiler
  → should only have to re-write front end

Targeting $M$ languages on $N$ machines:
- without IR: $M \times N$ compilers
- with IR: $M + N$ compilers
2. Speeding up an Interpreter

- Will be using library code a lot

- Don't want to have to load it in, do all the analysis (syntax and semantics) and all the translation

- Solution: pre-"compile" the libraries into IR

- Make your interpreter translate input into IR, then execute

- Also makes your interpreter more portable
## 3. Easing the Job for the Compiler/Interpreter

<table>
<thead>
<tr>
<th></th>
<th>High-Level Language</th>
<th>Assembly Language</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Types</strong></td>
<td>int, double, boolean, char,</td>
<td>byte, word...</td>
</tr>
<tr>
<td></td>
<td>String, BankAccount, ....</td>
<td></td>
</tr>
<tr>
<td><strong>Variables</strong></td>
<td>Local, static, private, pointers, ...</td>
<td>registers, memory</td>
</tr>
<tr>
<td><strong>Expressions</strong></td>
<td>2+3*4-Math.PI</td>
<td>addl %edx, (%eax)</td>
</tr>
<tr>
<td><strong>Control</strong></td>
<td>if, while, do...while, for, ....</td>
<td>test-and-jump</td>
</tr>
<tr>
<td><strong>Abstraction</strong></td>
<td>static/virtual functions/methods, pass by value/reference</td>
<td>subroutines, arguments on stack</td>
</tr>
<tr>
<td>** Extras:**</td>
<td>Inheritance, classes, structs, unions, ...</td>
<td>...?</td>
</tr>
</tbody>
</table>
Java’s Reasons for using an IR

- Embedded systems - wanted *portable* code
- Internet - want *mobile* code
- E-Commerce - want *secure* mobile code
- (?) Software Engineering - want *safe* code
- (?) SUN vs. MS - *really* need portable code
Java’s Architecture

• the Java Application Programming Interface (API)

• the Java programming language

• the Java class file format

• The Java Virtual Machine (JVM)
  = class loader + execution engine

“Java and all Java-based marks are trademarks or registered trademarks of Sun Microsystems, Inc. in the United States and other countries.”
Java’s IR - the .class file format

- Knows about classes, objects, methods
- Doesn’t have if-else/while - uses jumps
- Doesn’t use registers - \textbf{stack-based} machine
- Extra safeguards for security
- More later.....
JVM - Loading Classes

- There's a part of the JVM called the **Class Loader**

- The Class Loader supplies bytecodes (from .class files) to the bytecode interpreter

- You get one (trusted) "primordial" Class Loader

- You can create your own Class Loaders (with various levels of privileges)

- There's a class: `java.lang.ClassLoader`
JVM - The Execution Engine

Options:

- interpret everything
- compile everything
- just-in-time compiler (JIT) - compile as you run code the first time
- hotspot technology - compile “busy” parts of code