CMSC 430
Introduction to Compilers
Fall 2018

Language Virtual Machines

Introduction

- So far, we've focused on the compiler "front end"
  - Syntax (lexing/parsing)
  - High-level language semantics

- Ultimately, we want to generate code that runs our program on a "real" machine

- What machine should we target?
  - We could pick a specific hardware architecture
  - But we probably want our programs to run on multiple

- A common approach: target an abstracted machine, implement that machine for each real system

Virtual Machines

- Transform program into an intermediate representation (IR) with well-defined semantics
- Can *interpret* the IR using a *virtual machine*
  - Java, Lua, OCaml, .NET CLR, …
  - "Virtual" just means implemented in software, rather than hardware, but even hardware uses some interpretation
    - E.g., x86 processor has complex instruction set that's internally interpreted into much simpler form
- Alternatively, can use the IR as input for machine-specific compilation
  - LLVM
- Tradeoffs?

Java Virtual Machine (JVM)

- JVM memory model
  - Stack (function call frames, with local variables)
  - Heap (dynamically allocated memory, garbage collected)
  - Constants

- Bytecode files contain
  - Constant pool (shared constant data)
  - Set of classes with fields and methods
    - Methods contain instructions in Java bytecode language
    - Use javap -c to disassemble Java programs so you can look at their bytecode

JVM Semantics

- Documented in the form of a 600+ page PDF
  - https://docs.oracle.com/javase/specs/jvms/se11/jvms11.pdf

- Many concerns
  - Binary format of bytecode files
    - Including constant pool
  - Description of execution model (running individual instructions)
  - Java bytecode verifier
  - Thread model

JVM Design Goals

- Type- and memory-safe language
  - Mobile code—need safety and security
- Small file size
  - Constant pool to share constants
  - Each instruction is a byte (only 256 possible instructions)
- Good performance
- Good match to Java source code

JVM Execution Model

- From the JVM spec:
  - Virtual Machine Start-up
  - Loading
  - Linking: Verification, Preparation, and Resolution
  - Initialization
  - Detailed Initialization Procedure
  - Creation of New Class Instances
  - Finalization of Class Instances
  - Unloading of Classes and Interfaces
  - Virtual Machine Exit

JVM Instruction Set

- *Stack-based language*
  - Each thread has a private stack
  - All instructions take operands from the stack
- Categories of instructions
  - Load and store (e.g. aload_0,istore)
  - Arithmetic and logic (e.g. ladd,fcmpl)
  - Type conversion (e.g. i2b,d2i)
  - Object creation and manipulation (new,putfield)
  - Operand stack management (e.g. swap,dup2)
  - Control transfer (e.g. ifeq,goto)
  - Method invocation and return (e.g. invokespecial,areturn)

Example

```
public class hello {
   public static void main(String[] args) {
      System.out.println("Hello, world!");
   }
}
```

- Try compiling with javac, look at result using javap -c
- Things to look for:
  - Various instructions; references to classes, methods, and fields; exceptions; type information
- Things to think about:
  - File size really compact (Java → J)? Mapping onto machine instructions; performance; amount of abstraction in instructions

Other Languages

- While VMs provide convenient abstractions over physical machines, they can also be a target for multiple front-end languages

- Typically, also allows language interoperability

- The JVM has become a popular target
  - Scala, Kotlin, Clojure, Jython, JRuby, …

- Other VMs, such as the Microsoft .NET CLR, were designed as IRs for multiple languages
  - https://docs.microsoft.com/en-us/dotnet/standard/clr

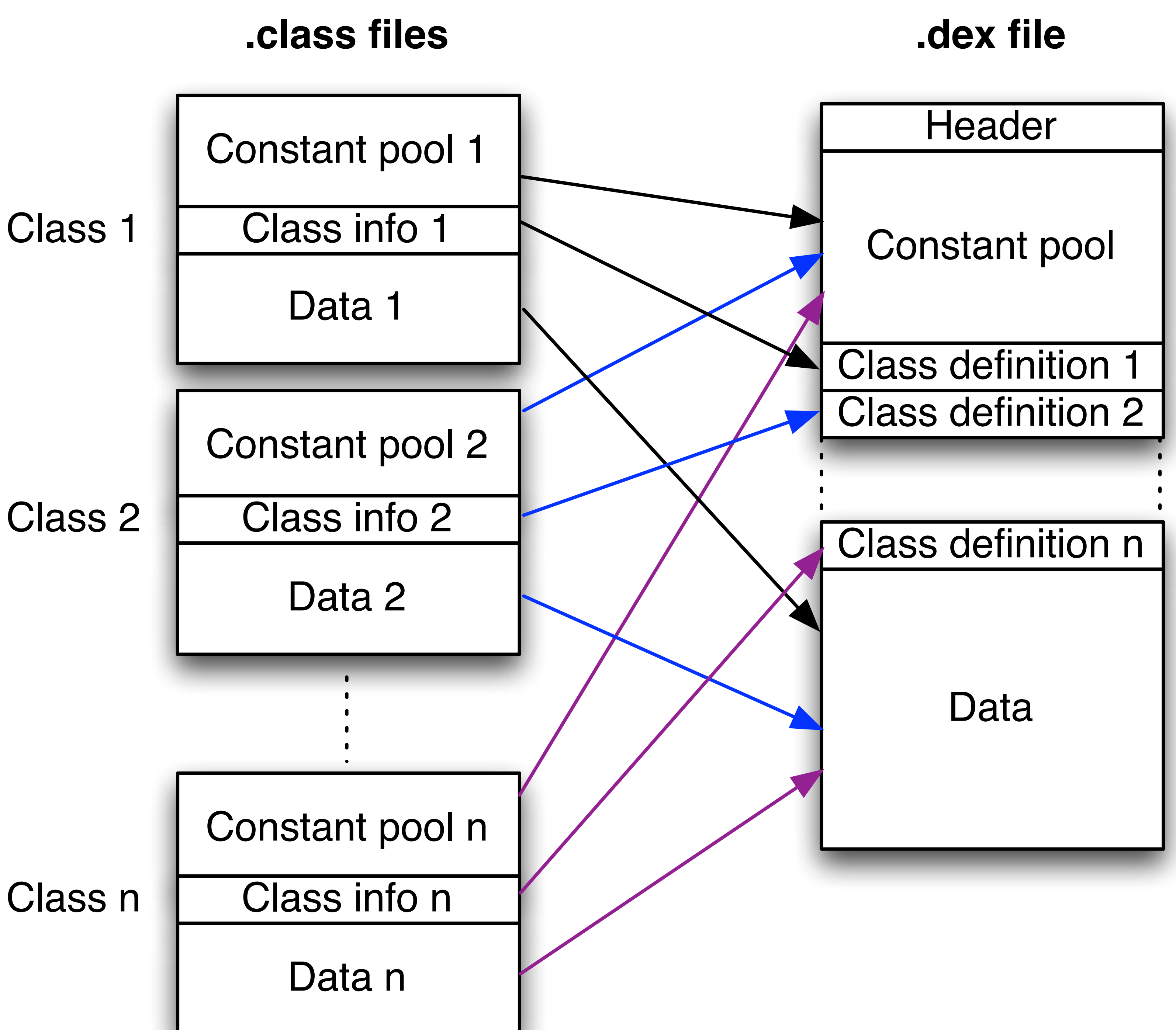
JVM Implementations

- There are many, particularly for embedded
  - https://en.wikipedia.org/wiki/List_of_Java_virtual_machines
- Sun (now Oracle) built the primary VM: **HotSpot**
  - Part of the JRE, OpenJDK
  - http://openjdk.java.net/groups/hotspot/
- Popular in the research community: **Jikes**
  - Implemented in Java (“metacircular”)
  - https://www.jikesrvm.org/

Dalvik Virtual Machine

- Alternative target for Java
- Developed by Google for Android phones
  - Register-, rather than stack-, based
  - Designed to be even more compact
- .dex (Dalvik) files are part of apk’s that are installed on phones (apks are zip files, essentially)
  - All classes must be joined together in one big .dex file, contrast with Java where each class separate
  - .dex produced from .class files

Compiling to .dex



- Many .class files ⇒ one .dex file
- Enables more sharing

Source for this and several of the following slides:: Octeau, Enck, and McDaniel. The ded Decompiler. Networking and Security Research Center Tech Report NAS-TR-0140-2010, The Pennsylvania State University. May 2011. http://siis.cse.psu.edu/ded/papers/NAS-TR-0140-2010.pdf

Dalvik is Register-Based

```java
public int add(int a, int b)
{
    return a + b;
}
```

(a) Source Code

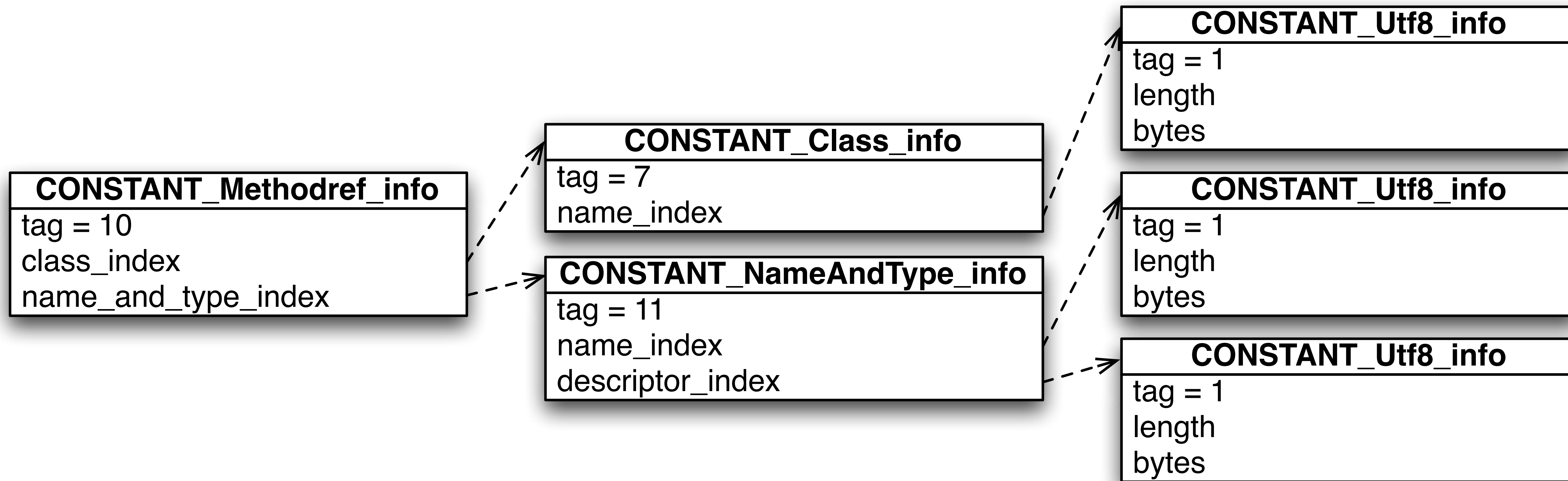
```
public int add(int, int)
   0:   iload_1
   1:   iload_2
   2:   iadd
   3:   ireturn
```

(b) Java (stack) bytecode

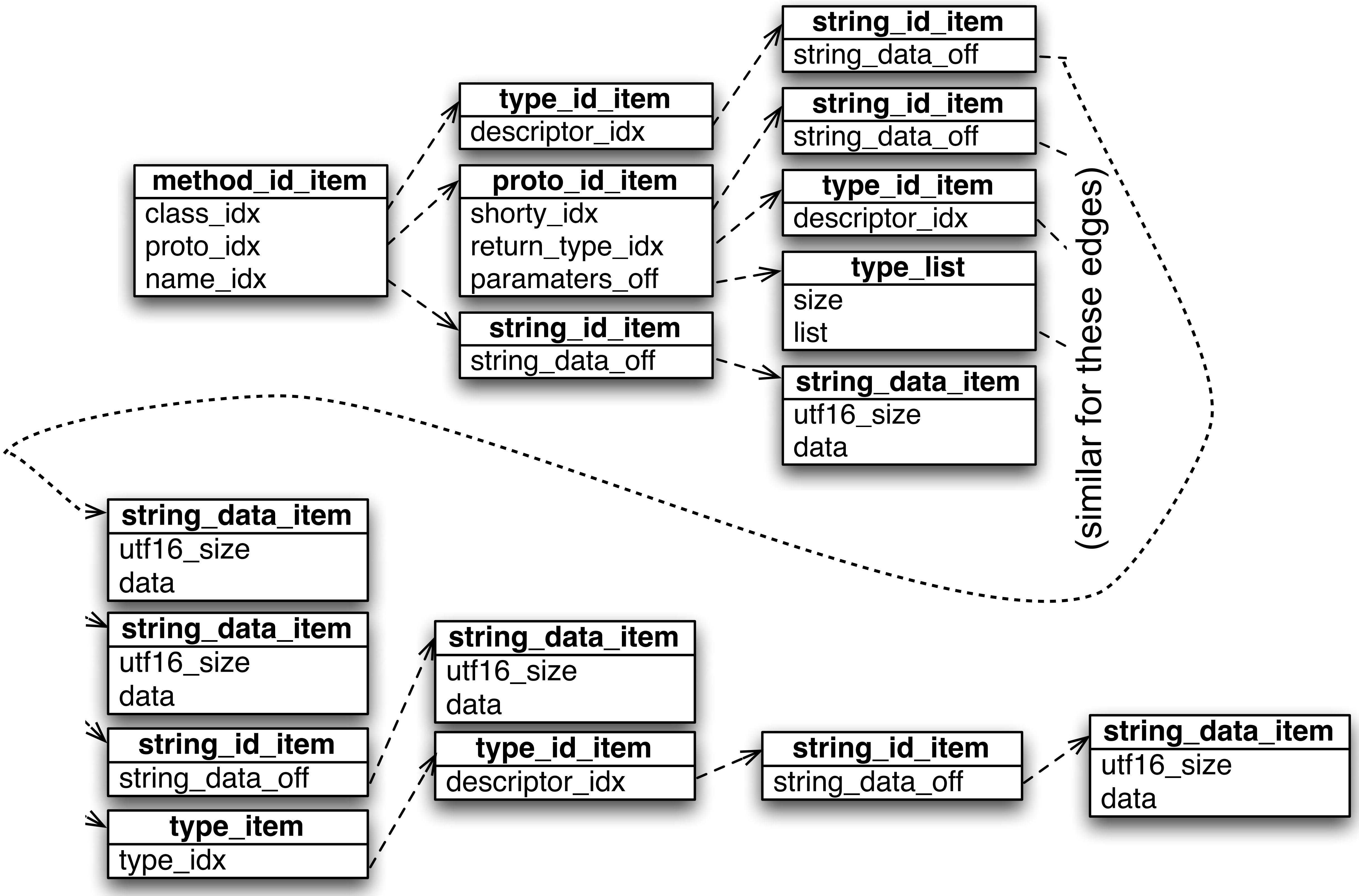
```
public int add(int, int)
   0:   add-int v0,v2,v3
   2:   return  v0
```

(c) Dalvik (register) bytecode

JVM Levels of Indirection



Dalvik Levels of Indirection



Discussion

- Why did Google invent its own VM?
  - Licensing fees? (now a settled lawsuit)
  - Performance?
  - Code size?
  - Anything else?

- Dalvik is no longer the primary runtime
  - Replaced by Android Runtime (ART)
  - https://source.android.com/devices/tech/dalvik

Just-in-time Compilation (JIT)

- Virtual machine that compiles some bytecode all the way to machine code for improved performance
  - Begin interpreting IR
  - Find performance critical sections
  - Compile those to native code
  - Jump to native code for those regions
- Tradeoffs?
  - Compilation time becomes part of execution time

Trace-Based JIT

- Used by HotSpot for Java
- Very popular for modern Javascript interpreters
  - JS hard to compile efficiently, because of large distance between its semantics and machine semantics
    - Many unknowns sabotage optimizations, e.g., in e.m(...), what method will be called?
- Idea: find a critical (often used) trace of a section of the program's execution, and compile that
  - Jump into the compiled code when hit beginning of trace
  - Need to be able to back out in case conditions for taking trace are not actually met

Project 3

- For project 3 you will implement your own small VM

- In OCaml, of course :)

- Simple machine model:
  - Functions with instructions
  - Heap: global variables
  - Stack with frames: caller, pc, registers
  - Unlimited registers

- Target for code generation in P4-P6