Languages and Compilers (SProg og Oversættere)

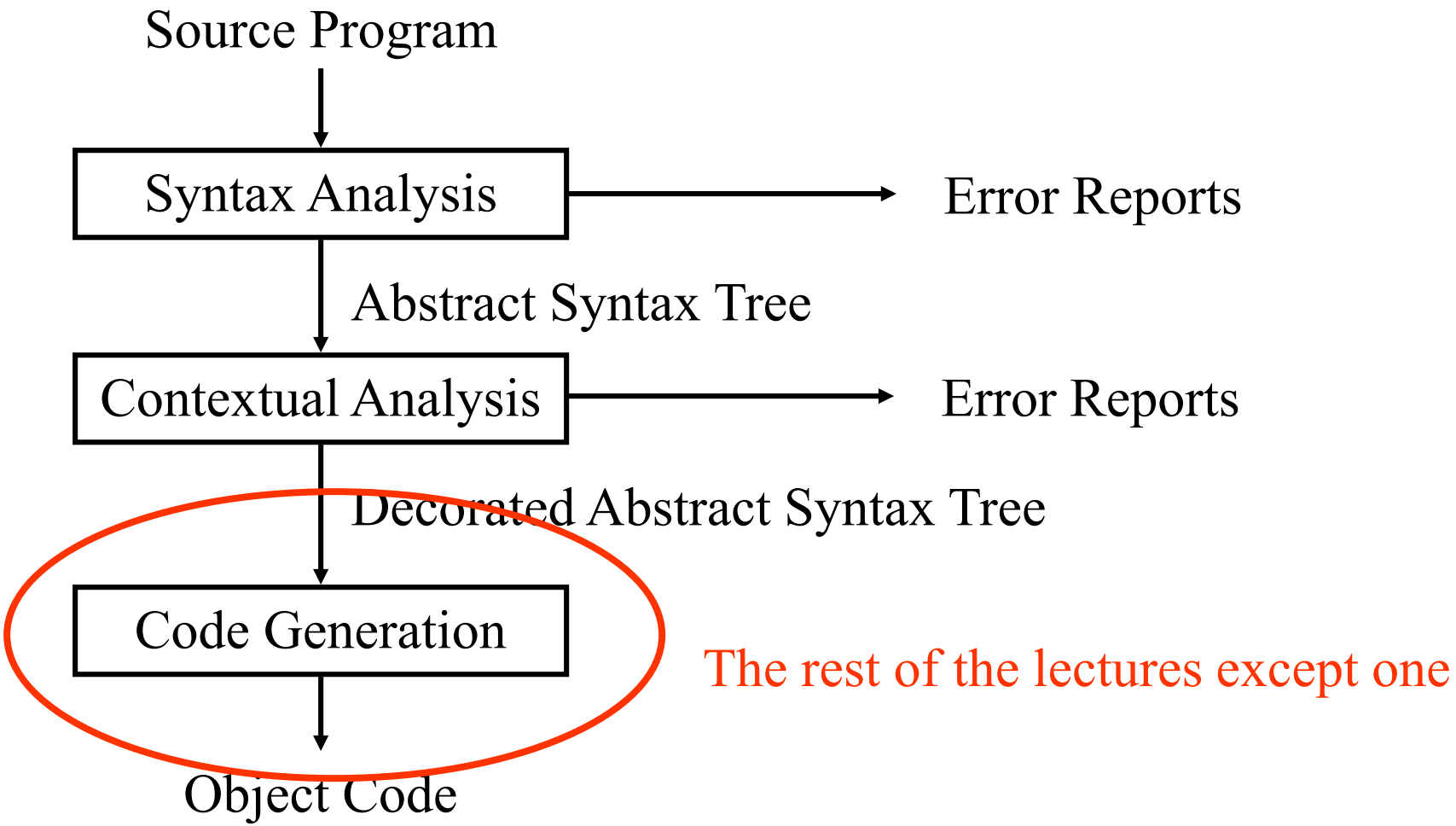
Lecture 15
Intermediate Representations

Bent Thomsen

Department of Computer Science

Aalborg University

The "Phases" of a Compiler



What's next?

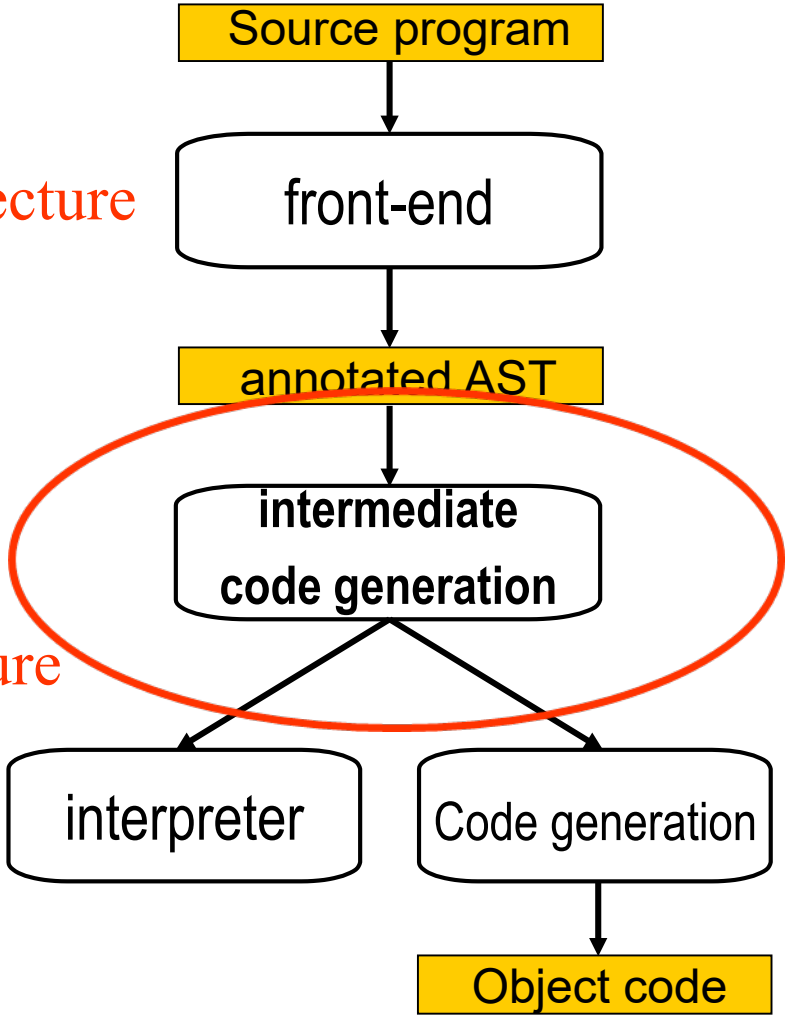
- interpretation
- intermediate code

Last lecture

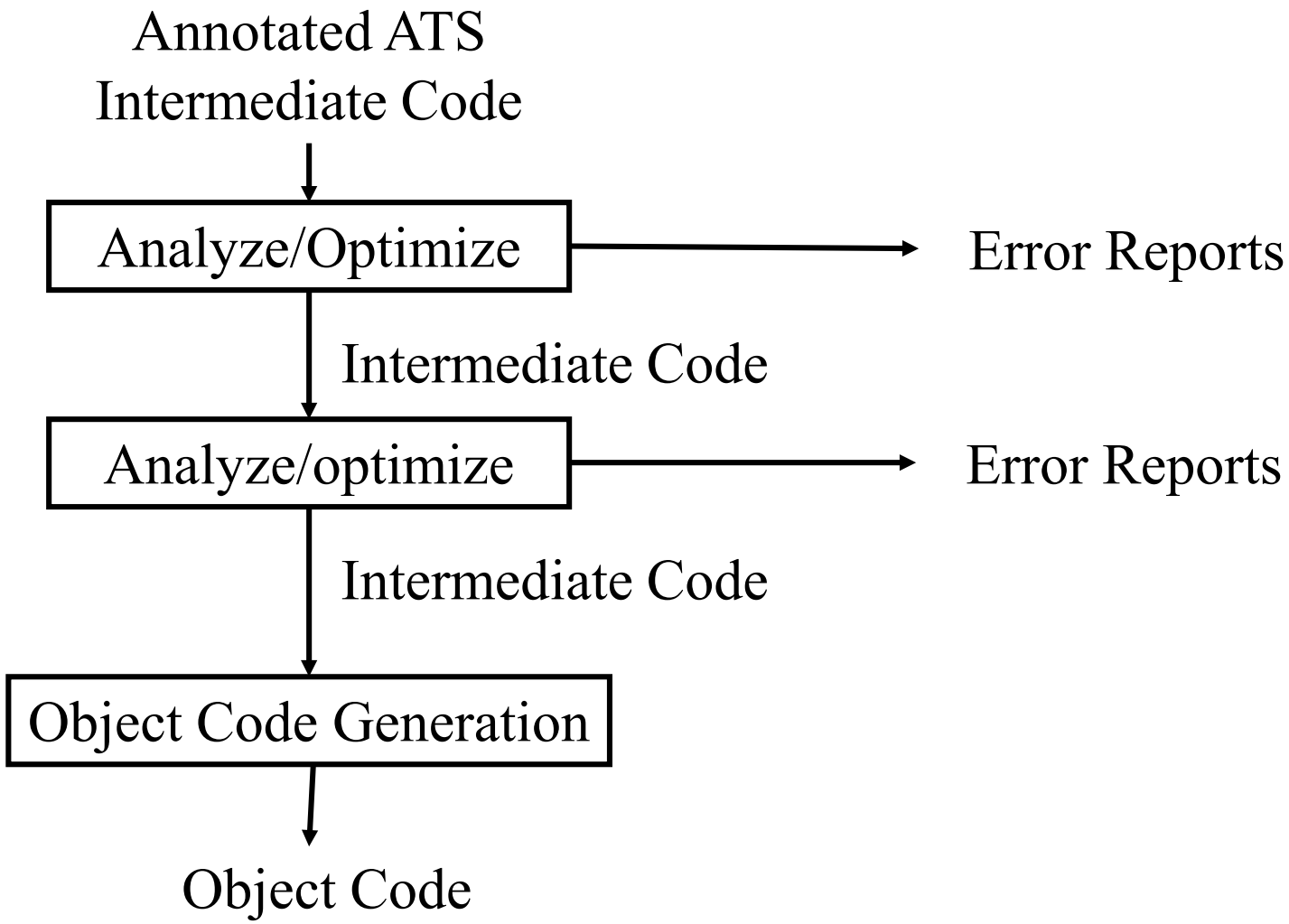
This lecture

- code generation
  - code selection
  - register allocation
  - instruction ordering

Next lecture



The Code generation "Phases" of a Compiler



Intermediate Representations

- Abstract Syntax Tree
  - Convenient for semantic analysis phases
  - Convenient for recursive interpretation
  - We can generate code directly from the AST, but...
  - What about multiple target architectures?
- Intermediate Representation
  - "Neutral" architecture
  - Easy to translate to native code
  - Can abstracts away complicated runtime issues
    - Stack Frame Management
    - Memory Management
    - Register Allocation

Overview

- Semantic gap between high-level source languages and target machine language
- Examples
  - Early C++ compilers
    - cpp: preprocessor
    - cfront: translate C++ into C
    - C compiler

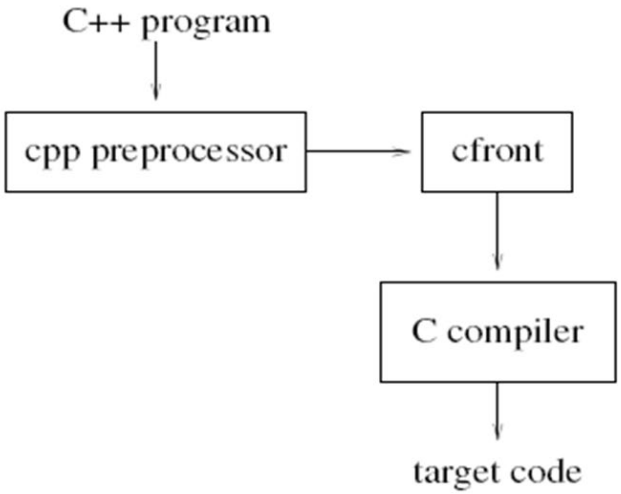


Figure 10.1: Use of `cfront` to translate C++ to C.

Another Example

- LaTeX
  - TeX: designed by Donald Knuth
  - dvi: device-independent intermediate representation
  - Ps: PostScript
  - pixels
- Portability enhanced

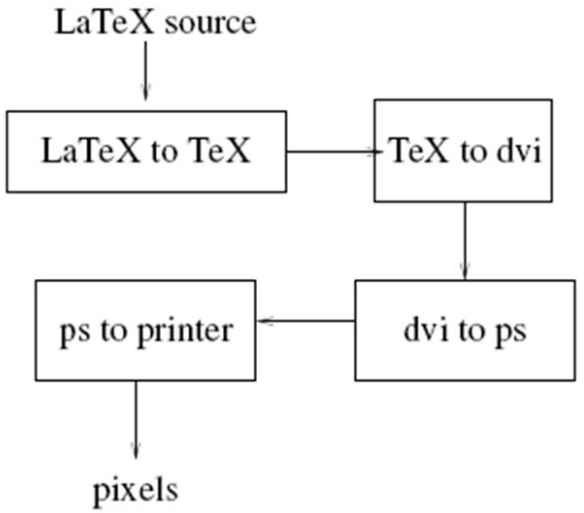


Figure 10.2: Translation from LaTeX into print.

Challenges

- Challenges
  - An intermediate language (IL) must be precisely defined
  - Translators and processors must be crafted for an IL
  - Connections must be made between levels so that feedback from intermediate steps can be related to the source program
- Other concerns
  - Efficiency

- Compiler suites that host multiple source languages and target multiple instruction sets obtain great leverage from a middle-end
  - Ex: s source languages, t target languages
    - s*t vs. s+t
-

s*t vs. s+t

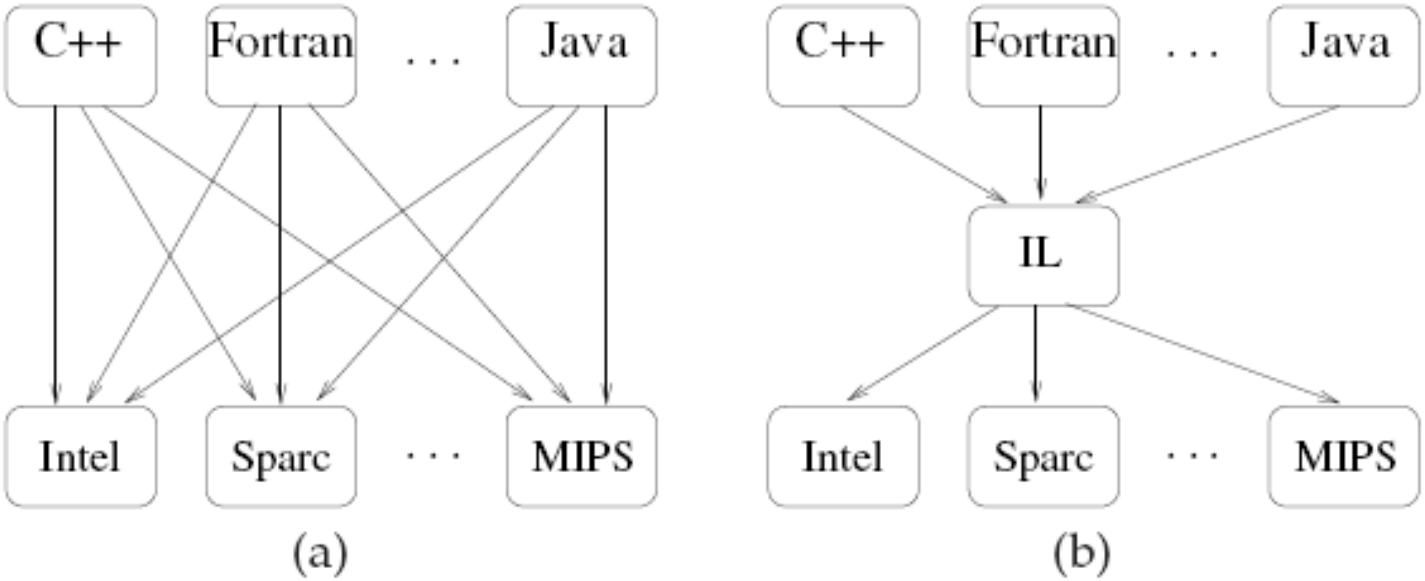


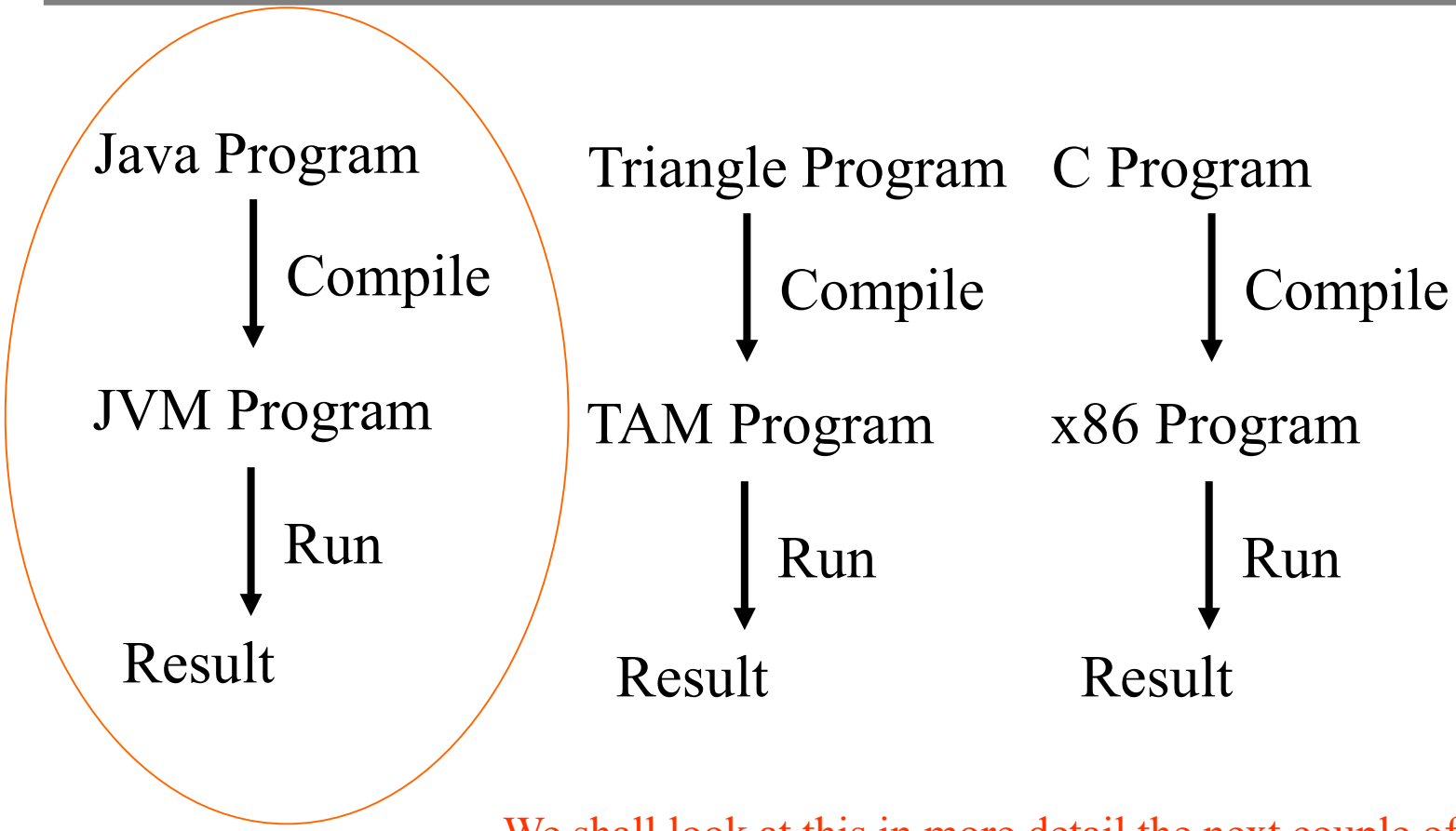
Figure 10.3: A middle-end and its ILs simplify construction of a compiler suite that must support multiple source languages and multiple target architectures.

IL Advantages

- An IL simplifies development and testing of system components
  - simplify the pioneering and prototyping of news ideas
- An IL allows various system components to interoperate by facilitating access to information about the program
  - E.g. variable names and types, and source line numbers could be useful in the debugger
  - It allows components and tools to interface with other products
- An IL enables the crafting of a retargetable code generator, which greatly enhances its portability
  - Pascal: P-code
  - Ada: DIANA (Descriptive Intermediate Attributed Notation for Ada)
  - C: RTL
  - Java: JVM
  - C#: CIL
  - Python: Python Byte Code

Code Generation

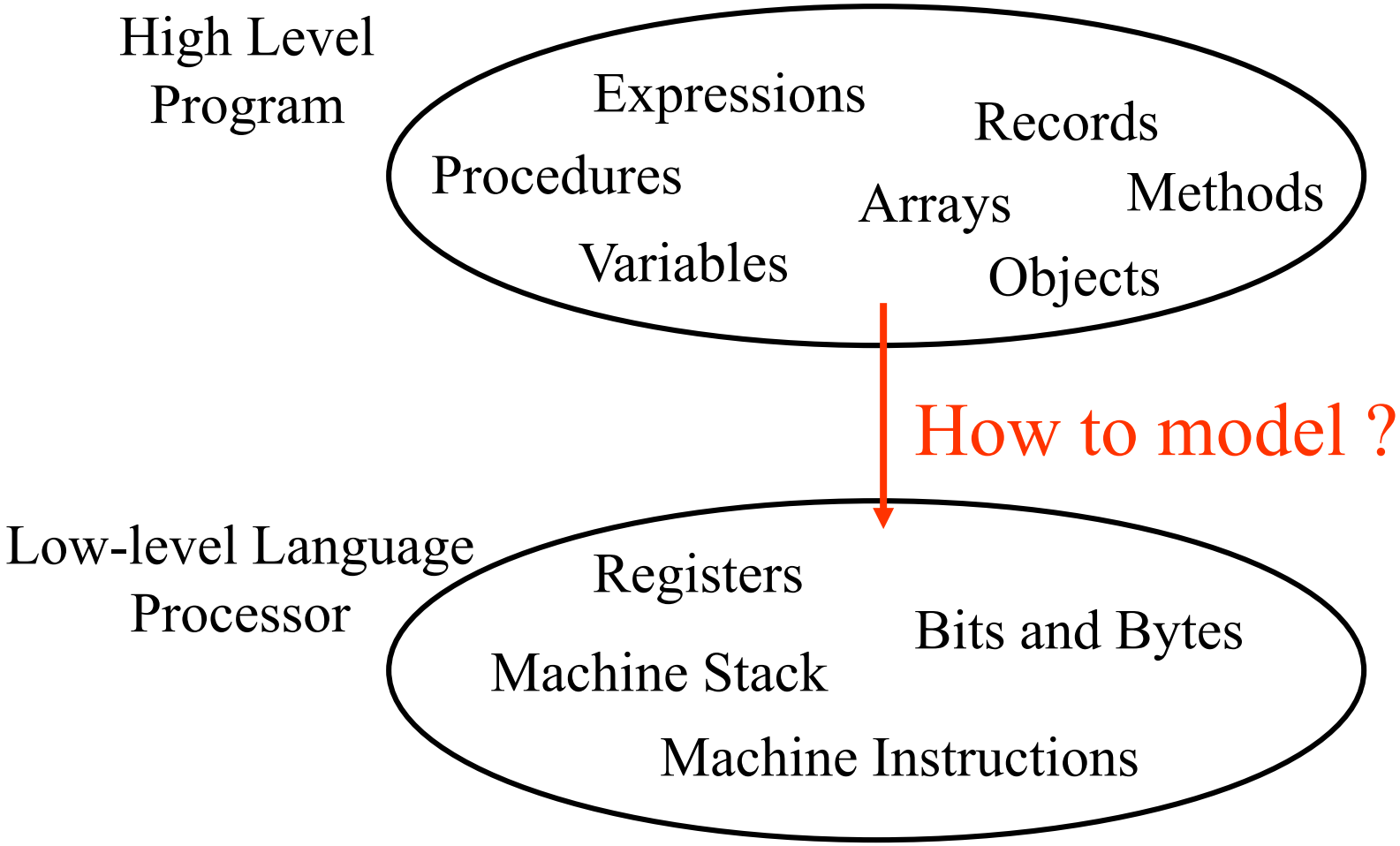
A compiler translates a program from a high-level language into an **equivalent** program in a low-level language.



We shall look at this in more detail the next couple of lectures
Note that code generation is specific to the target, but we try to generalize

What are (some of) the issues

How to model high-level computational structures and data structures in terms of low-level memory and machine instructions.



Easy for Java (or Java like) on the JVM

Type	JVM designation
`boolean`	`Z`
`byte`	`B`
`double`	`D`
`float`	`F`
`int`	`I`
`long`	`J`
`short`	`S`
`void`	`V`
Reference type *t*	`L`*t*`;`
Array of type *a*	`[`*a*

Figure 10.4: Java types and their designation in the JVM. All of the integer-valued types are signed. For reference types, *t* is a fully qualified class name. For array types, *a* can be a primitive, reference, or array type.

For other Languages on the JVM some thoughts
Are needed on a suitable mapping

The JVM

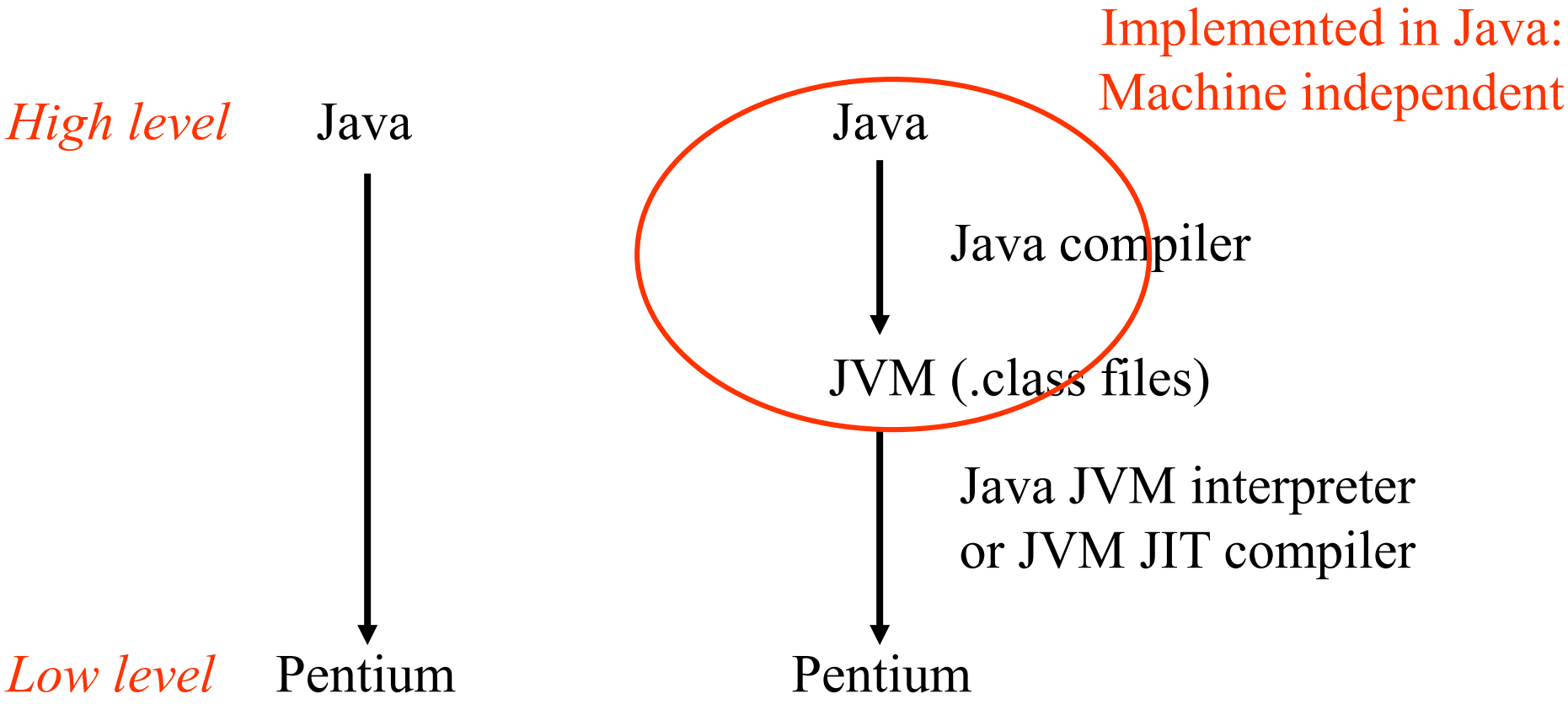
We now look at the JVM as an example of a real-world runtime system for a modern object-oriented programming language.

The material in this lecture is interesting because:

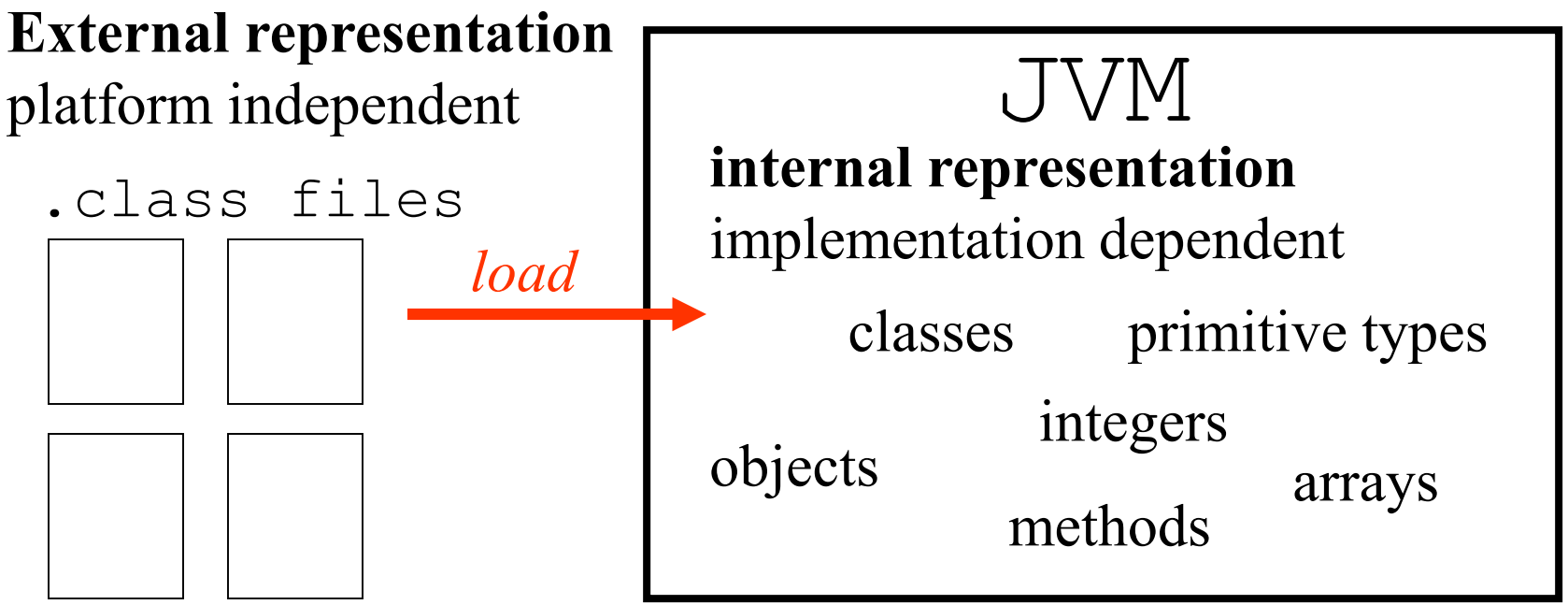
1) it will help understand some things about the JVM
2) JVM is probably the most common and widely used VM in the world.
3) You'll get a better idea what a real VM looks like.
4) You may choose the JVM as a target for your own compiler

Abstract Machines

An abstract machine implements an intermediate language "in between" the high-level language (e.g. Java) and the low-level hardware (e.g. Pentium)



Class Files and Class File Format



The **JVM is** an **abstract** machine in the true sense of the word.

The JVM spec. does not specify implementation details (can be dependent on target OS/platform, performance requirements etc.)

The JVM spec defines a machine independent "**class file format**" that all JVM implementations must support.

Java Virtual Machine

- Class files:
  - binary encodings of the data and instructions in a Java program
- Design principles
  - Compactness
    - Instructions in nearly zero-address form
- Class file contains:
  - Table of constants.
  - Tables describing the class
    - name, superclass, interfaces
    - attributes, constructor
  - Tables describing fields and methods
    - name, type/signature
    - attributes (private, public, etc)
  - The code for methods.

```
ClassFile {
      u4 magic; //always (0xCAFEBABE)
      u2 minor_version;
      u2 major_version;
      u2 constant_pool_count;
      cp_info constant_pool[constant_pool_count-1];
      u2 access_flags;
      u2 this_class;
      u2 super_class;
      u2 interfaces_count;
      u2 interfaces[interfaces_count];
      u2 fields_count;
      field_info fields[fields_count];
      u2 methods_count;
      method_info methods[methods_count];
      u2 attributes_count;
      attribute_info attributes[attributes_count];
   }
```

Data Types

JVM (and Java) distinguishes between two kinds of types:

Primitive types:

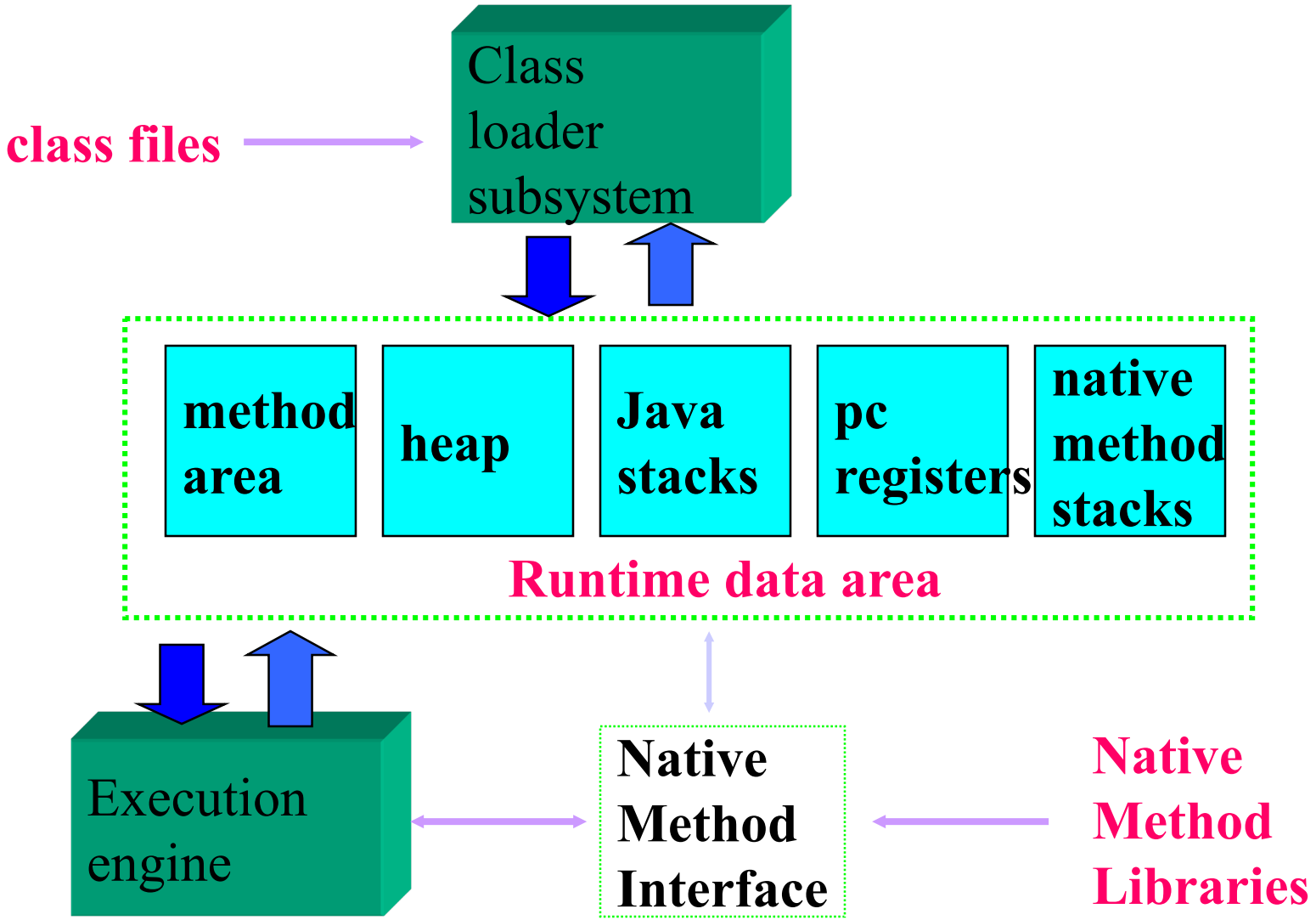
- boolean: `boolean`
- numeric integral: `byte, short, int, long, char`
- numeric floating point: `float, double`
- internal, for exception handling: `returnAddress`
    - `Used by jsr, jsr_w, ret instructions`

Reference types:

- class types
- array types
- interface types

Note: Primitive types are represented directly, reference types are represented indirectly (as pointers to array or class instances)

Internal Architecture of JVM



Class Loading

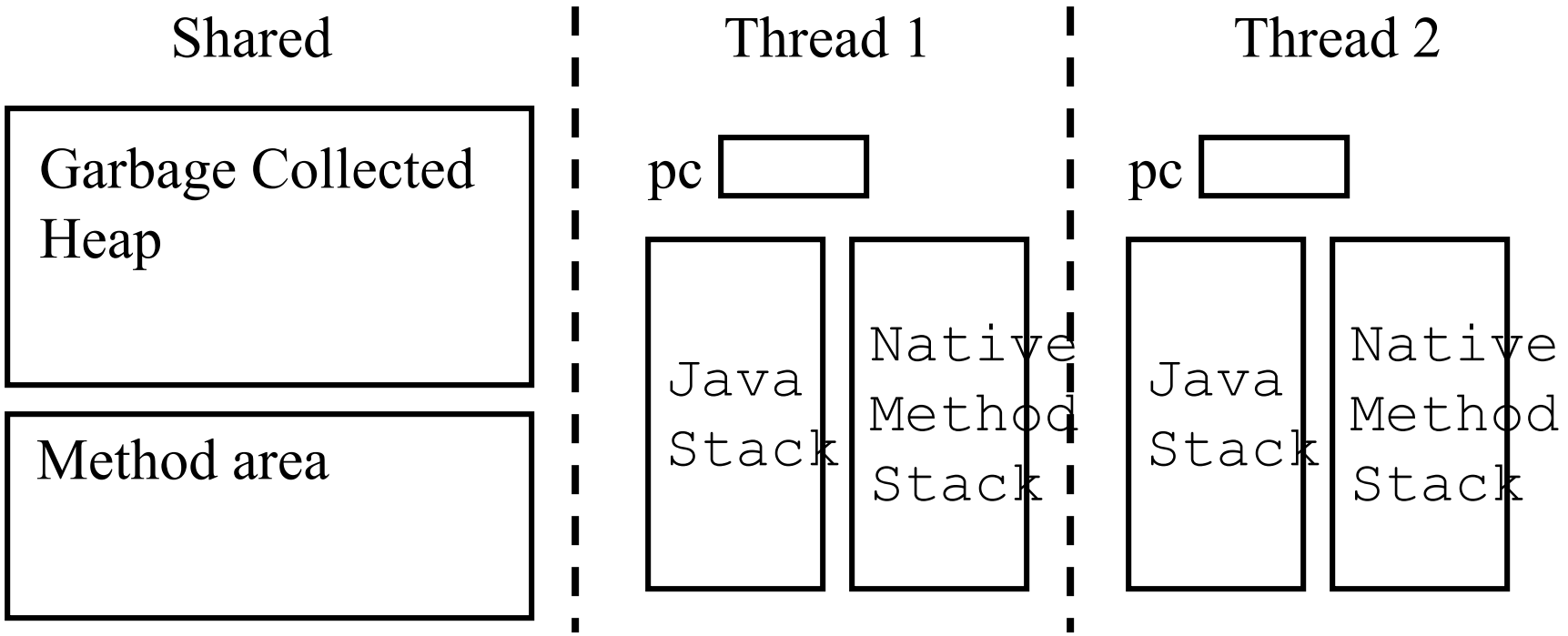
- Classes are loaded lazily when first accessed
  - Though some JVMs do eager loading
- Class name must match file name
- Super classes are loaded first (transitively)
- The bytecode is verified
- Static fields are allocated and given default values
- Static initializers are executed

JVM: Runtime Data Areas

Besides OO concepts, JVM also supports multi-threading. Threads are directly supported by the JVM.

=> Two kinds of runtime data areas:

1) shared between all threads
2) private to a single thread



Java Stacks

JVM is a stack based machine

JVM instructions

- implicitly take arguments from the stack top
- put their result on the top of the stack

The stack is used to

- pass arguments to methods
- return result from a method
- store intermediate results in evaluating expressions
- store local variables

Expression Evaluation on a Stack Machine

Example 1: Computing `(a * b) + (1 - (c * 2))` on a stack machine.

```
LOAD a              //stack: a
LOAD b              //stack: a b
MULT                //stack: (a*b)
LOAD #1             //stack: (a*b) 1
LOAD c              //stack: (a*b) 1 c
LOAD #2             //stack: (a*b) 1 c 2
MULT                //stack: (a*b) 1 (c*2)
SUB                 //stack: (a*b) (1-(c*2))
ADD                 //stack: (a*b)+(1-(c*2))
```

Note the correspondence between the instructions and the expression written in postfix notation: `a b * 1 c 2 * - +`

Expression Evaluation on a Stack Machine

Example 2: Computing `(0 < n) && odd(n)` on a stack machine.

```
LOAD #0          //stack: 0
LOAD n           //stack: 0 n
LT               //stack: (0<n)
LOAD n           //stack: (0<n) n
CALL odd         //stack: (0<n) odd(n)
AND              //stack: (0<n)&&odd(n)
```

This example illustrates that calling functions/procedures fits in just as naturally with the stack machine evaluation model as operations that correspond to machine instructions.

In register machines this is much more complicated, because a stack must be created in memory for managing subroutine calls/returns.

JVM Interpreter

The core of a JVM interpreter is basically this:

```
do {
    byte opcode = fetch an opcode;
    switch (opcode) {
      case opCode1 :
          fetch operands for opCode1;
          execute action for opCode1;
          break;
      case opCode2 :
          fetch operands for opCode2;
          execute action for opCode2;
          break;
      case ...
} while (more to do)
```

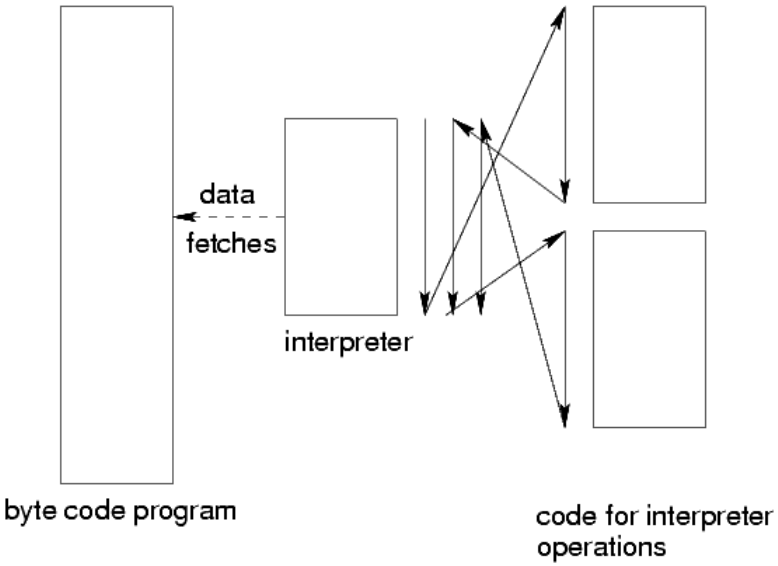
The JVM interpreter loop in the HVM

```
1 static int32 methodInterpreter(const
      MethodInfo* method, int32* fp) {
2   unsigned char *method_code;
3   int32* sp;
4   const MethodInfo* methodInfo;
5
6   start: method_code = (unsigned char *)
        pgm_read_pointer(&method->code, unsigned
         char**);
7   sp = &fp[pgm_read_word(&method->maxLocals)
        +2];
8
9   loop: while (1) {
10    unsigned char code = pgm_read_byte(
          method_code);
11    switch (code) {
12    case ICONST_0_OPCODE:
13    //ICONST_X Java Bytecodes
14    case ICONST_5_OPCODE:
15      *sp++ = code - ICONST_0_OPCODE;
16      method_code++;
17      continue;
18    case FCONST_0_OPCODE:
19    //Remaining Java Bytecode impl...
20    }
21  }
22 }
```

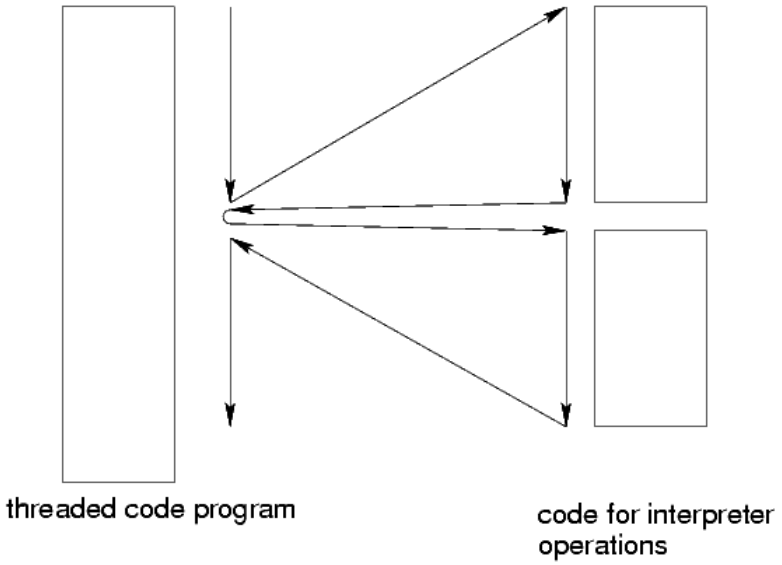
Threaded Code

Switch-Cased statement often translated into a jump table
Indexing through a jump table is expensive.
Idea: Use the address of the code for an operation as the opcode for that operation.

byte code:



threaded code:



Control flow behavior:

[based on: James R. Bell. Threaded Code. *Communications of the ACM*, vol. 16 no. 6, June 1973, pp. 370–372]

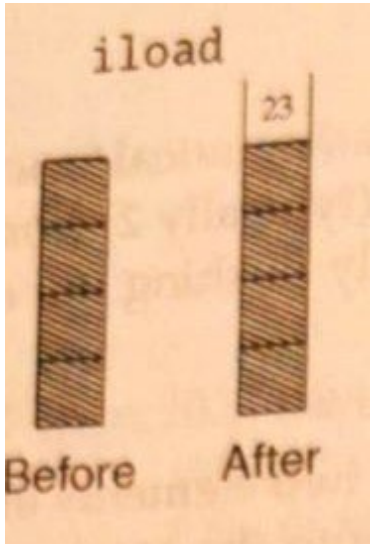
Instruction-set: typed instructions!

JVM instructions are explicitly typed: different opCodes for instructions for integers, floats, arrays and reference types.

This is reflected by a naming convention in the first letter of the opCode mnemonics:

Example: different types of "load" instructions

`iload`	integer load
`lload`	long load
`fload`	float load
`dload`	double load
`aload`	reference-type load



Instruction set: kinds of operands

JVM instructions have three kinds of operands:

- from the top of the operand stack
- from the bytes following the opCode
- part of the opCode

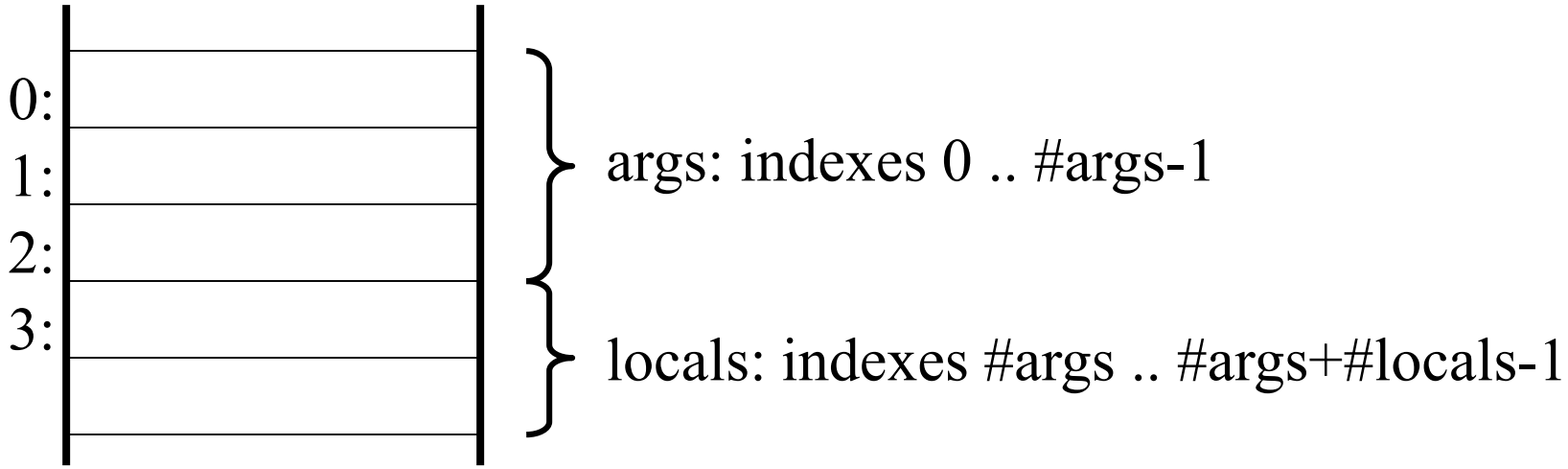
One instructions may have different "forms" supporting different kinds of operands.

Example: different forms of "iload".

Assembly code	Binary instruction code layout		
`iload_0`	26		
`iload_1`	27		
`iload_2`	28		
`iload_3`	29		
`iload` *n*	21	*n*	
`wide iload` *n*	196	21	*n*

Instruction-set: accessing arguments and locals

arguments and locals area inside a stack frame



Instruction examples:

iload_1	istore_1
iload_3	astore_1
aload 5	fstore_3
aload_0	

- A load instruction: loads something from the args/locals area to the top of the operand stack.
- A store instruction takes something from the top of the operand stack and stores it in the argument/local area

Instruction-set: non-local memory access

In the JVM, the contents of different "kinds" of memory can be accessed by different kinds of instructions.

accessing locals and arguments: `load` and `store` instructions

accessing fields in objects: `getfield, putfield`

accessing static fields: `getstatic, putstatic`

Note: static fields are a lot like global variables. They are allocated in the "method area" where also code for methods and representations for classes are stored.

Q: what memory area are getfield and putfield accessing?

Instruction-set: operations on numbers

Arithmetic

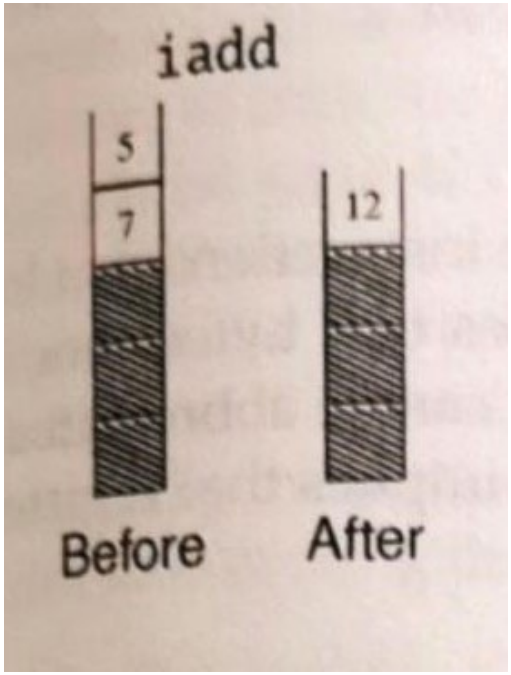
add: `iadd, ladd, fadd, dadd`
subtract: `isub, lsub, fsub, dsub`
multiply: `imul, lmul, fmul, dmul`

…

Conversion

`i2l, i2f, i2d`
`l2f, l2d, f2s`

`f2i, d2i, …`



Instruction-set …

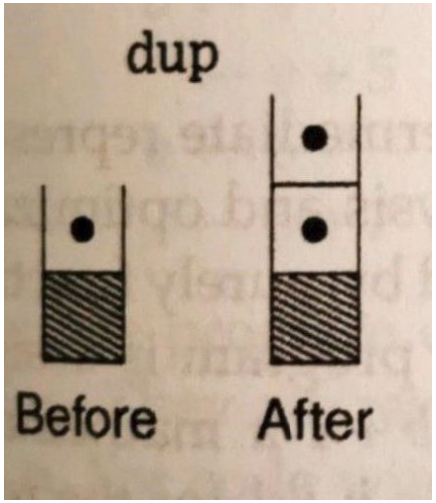
Operand stack manipulation

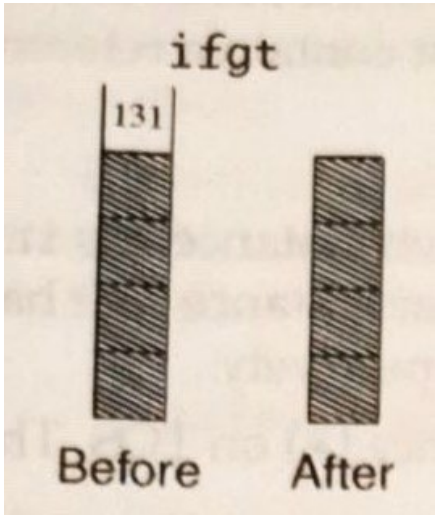
`pop, pop2, dup, dup2, dup_x1, swap, …`

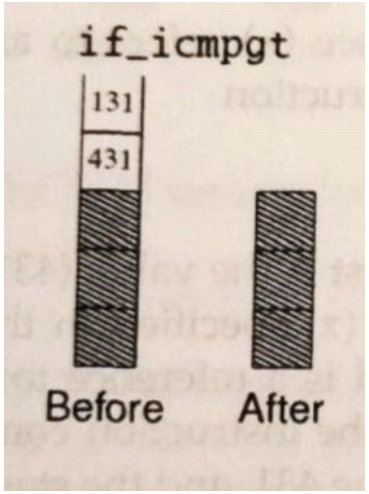
Control transfer

Unconditional : `goto, goto_w, jsr, ret, …`

Conditional: `ifeq, iflt, ifgt, …`







Instruction-set …

Method invocation:

`invokevirtual`

usual instruction for calling a method on an object.

`invokeinterface`

same as invokevirtual, but used when the called method is declared in an interface. (requires different kind of method lookup)

`invokespecial`

for calling things such as constructors. These are not dynamically dispatched (also known as `invokenonvirtual`)

`invokestatic`

for calling methods that have the "static" modifier (these methods "belong" to a class, rather an object)

Returning from methods:

`return, ireturn, lreturn, areturn, freturn, …`

Instruction-set: Heap Memory Allocation

Create new class instance (object):

new

Create new array:

newarray

for creating arrays of primitive types.

anewarray, multianewarray

for arrays of reference types

Example

As an example on the JVM, we will take a look at the compiled code of the following simple Java class declaration.

```
class Factorial {

    int fac(int n) {
        int result = 1;
        for (int i=2; i<n; i++) {
            result = result * i;
        }
        return result;
    }
}
```

Compiling and Disassembling

```
% javac Factorial.java
% javap -c -verbose Factorial
Compiled from Factorial.java
public class Factorial extends java.lang.Object {
    public Factorial();
        /* Stack=1, Locals=1, Args_size=1 */
    public int fac(int);
        /* Stack=2, Locals=4, Args_size=2 */
}

Method Factorial()
   0 aload_0
   1 invokespecial #1 <Method java.lang.Object()>
   4 return
```

Compiling and Disassembling ...

```
                    // address: 0    1 2      3
Method int fac(int) // stack: this n result i
  0 iconst_1        // stack: this n result i 1
  1 istore_2        // stack: this n result i
  2 iconst_2        // stack: this n result i 2
  3 istore_3        // stack: this n result i
  4 goto 14
  7 iload_2         // stack: this n result i result
  8 iload_3         // stack: this n result i result i
  9 imul            // stack: this n result i result i
 10 istore_2
 11 iinc 3 1
 14 iload_3         // stack: this n result i i
 15 iload_1         // stack: this n result i i n
 16 if_icmple 7     // stack: this n result i
 19 iload_2         // stack: this n result i result
 20 ireturn
```

JASMIN

- JASMIN is an assembler for the JVM
  - Takes an ASCII description of a Java class
  - Input written in a simple assembler like syntax
    - Using the JVM instruction set
  - Outputs binary class file
  - Suitable for loading by the JVM

- Running JASMIN
  - `jasmin myfile.j`
- Produces a .class file with the name specified by the .class directive in myfile.j

Example: out.j

```
.class public out
.super java/lang/Object

.method public <init>()V
        aload_0
        invokespecial java/lang/Object/<init>()V
        return
.end method

.method public static main([Ljava/lang/String;)V
        .limit stack 2

        getstatic java/lang/System/out Ljava/io/PrintStream;
        ldc “Hello World”
        invokevirtual java/io/PrintStream/println(Ljava/lang/String;)V

        return
.end method
```

The result: out.class

```
00000000h: CA FE BA BE 00 03 00 2D 00 1B 0C 00 17 00 1A 01 ; 栲瑣...-........
00000010h: 00 16 28 5B 4C 6A 61 76 61 2F 6C 61 6E 67 2F 53 ; ..([Ljava/lang/S
00000020h: 74 72 69 6E 67 3B 29 56 01 00 10 6A 61 76 61 2F ; tring;)V...java/
00000030h: 6C 61 6E 67 2F 4F 62 6A 65 63 74 01 00 06 3C 69 ; lang/Object...<i
00000040h: 6E 69 74 3E 07 00 03 0C 00 04 00 08 07 00 10 01 ; nit>............
00000050h: 00 03 28 29 56 07 00 13 01 00 04 43 6F 64 65 01 ; ..()V......Code.
00000060h: 00 04 6D 61 69 6E 09 00 09 00 01 01 00 0A 53 6F ; ..main........So
00000070h: 75 72 63 65 46 69 6C 65 01 00 05 6F 75 74 2E 6A ; urceFile...out.j
00000080h: 0C 00 11 00 19 01 00 13 6A 61 76 61 2F 69 6F 2F ; ........java/io/
00000090h: 50 72 69 6E 74 53 74 72 65 61 6D 01 00 07 70 72 ; PrintStream...pr
000000a0h: 69 6E 74 6C 6E 0A 00 05 00 06 01 00 10 6A 61 76 ; intln........jav
000000b0h: 61 2F 6C 61 6E 67 2F 53 79 73 74 65 6D 01 00 0B ; a/lang/System...
000000c0h: 48 65 6C 6C 6F 20 57 6F 72 6C 64 0A 00 07 00 0F ; Hello World.....
000000d0h: 08 00 14 01 00 03 6F 75 74 07 00 17 01 00 15 28 ; ......out......(
000000e0h: 4C 6A 61 76 61 2F 6C 61 6E 67 2F 53 74 72 69 6E ; Ljava/lang/Strin
000000f0h: 67 3B 29 56 01 00 15 4C 6A 61 76 61 2F 69 6F 2F ; g;)V...Ljava/io/
00000100h: 50 72 69 6E 74 53 74 72 65 61 6D 3B 00 21 00 18 ; PrintStream;.!..
00000110h: 00 05 00 00 00 00 00 02 00 01 00 04 00 08 00 01 ; ................
00000120h: 00 0A 00 00 00 11 00 01 00 01 00 00 00 05 2A B7 ; ..............*?
00000130h: 00 12 B1 00 00 00 00 00 09 00 0B 00 02 00 01 00 ; ..?.............
00000140h: 0A 00 00 00 15 00 02 00 01 00 00 00 09 B2 00 0C ; .............?.
00000150h: 12 16 B6 00 15 B1 00 00 00 00 00 01 00 0D 00 00 ; ..?.?.........
00000160h: 00 02 00 0E                                     ; ....
```

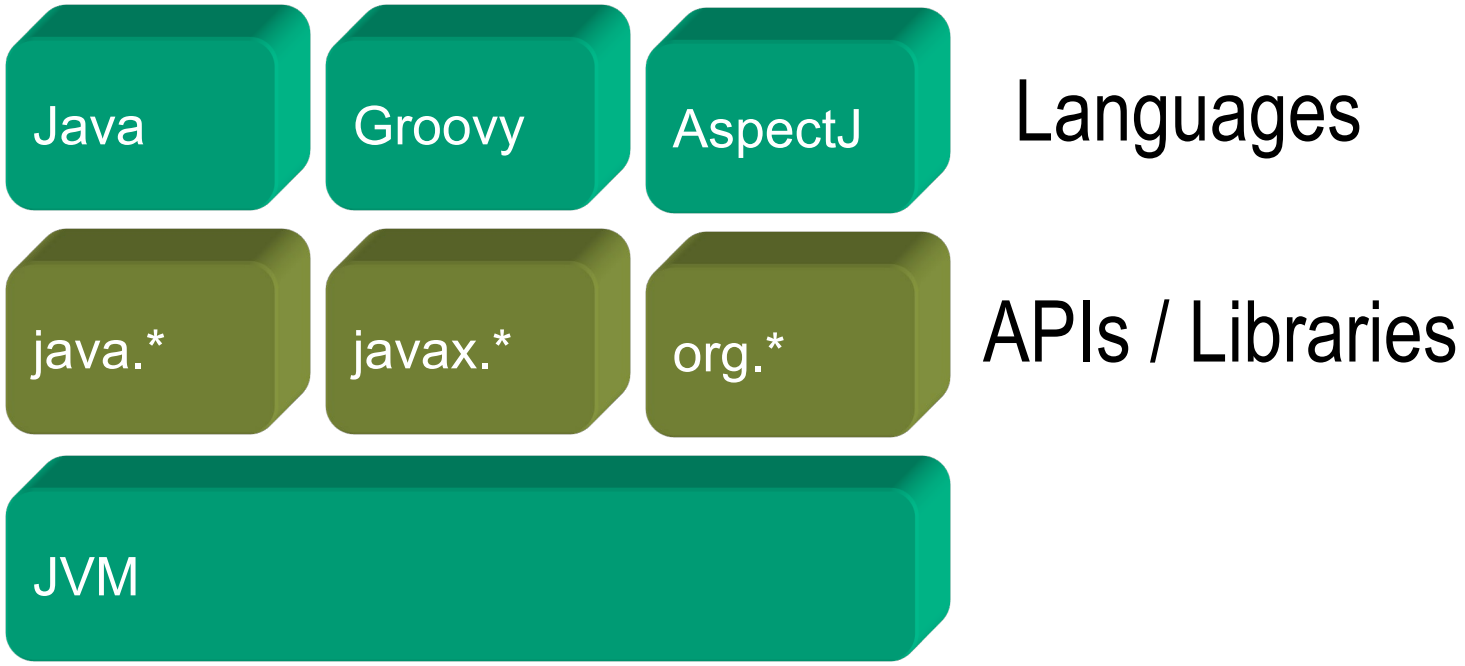
Jasmin file format

- Directives
  - .catch . Class .end .field .implements .interface .limit .line
  - .method .source .super .throws .var
- Instructions
  - JVM instructions: ldc, iinc bipush
- Labels
  - Any name followed by : - e.g. Foo:
  - Cannot start with = : . *
  - Labels can only be used within method definitions

The JVM as a target for different languages

When we talk about Java what do we mean?

- "Java" isn't just a language, *it is a platform*
- *The list of languages targeting the JVM is very long!*
  - *(Fortress), Scala, Clojure, Kotlin are currently very hot*
  - *http://en.wikipedia.org/wiki/List_of_JVM_languages*



Reusability

- Java has a lot of APIs and libraries
  - Core libraries (java[x].*)
  - Open source libraries
  - Third party commercial libraries

- What is it that we are *reusing* when we use these tools?
  - We are reusing the bytecode
  - We are reusing the fact that the JVM has a nice spec

- This means that we can innovate on top of this binary class file nonsense ☺

Not just one JVM, but a whole family

- JVM (J2EE & J2SE)
  - SUN Classis, SUN HotSpots, Oracle, IBM, BEA, …
- CVM, KVM (J2ME)
  - Small devices.
  - Reduces some VM features to fit resource-constrained devices.
- JCVM (Java Card)
  - Smart cards.
  - It has least VM features.

- And there are also lots of other JVMs
  - E.g. HVM (www.icelab.dk)

Java Platform & VM & Devices

Java Technology Targets a Broad Range of Devices

Hardware implementations of the JVM

Figure 6. aJ-100 package (larger than actual size).

Pause

s*t vs. s+t

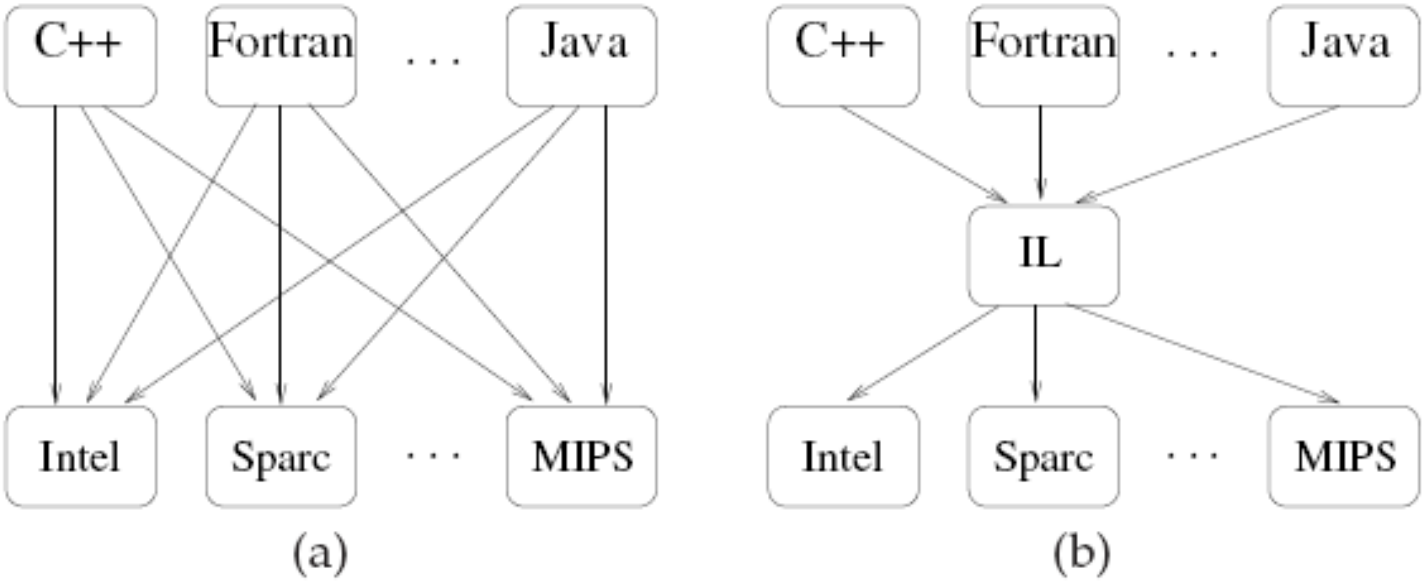
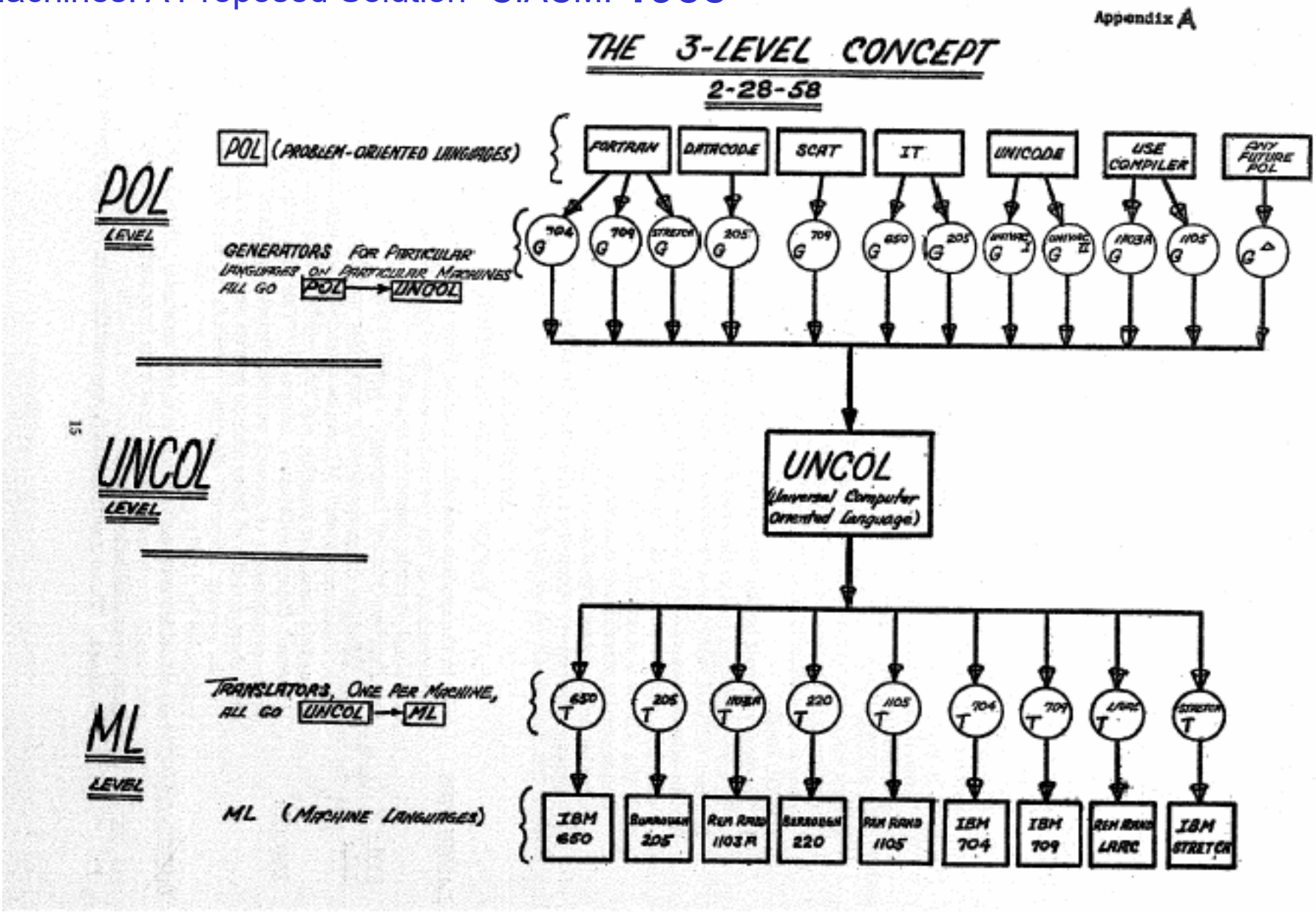


Figure 10.3: A middle-end and its ILs simplify construction of a compiler suite that must support multiple source languages and multiple target architectures.

The common intermediate format nirvana

- If we have `n` languages and need to have them running on `m` machines we need `m*n` compilers!
- If we have one common intermediate format we only need `n` front-ends and m back-ends, i.e. `m+n`
- ”Why haven’t you taught us about the common intermediate language?”

Strong et al. “The Problem of Programming Communication with Changing Machines: A Proposed Solution” C.ACM. **1958**



Quote

This concept is not particularly new or original. It has been discussed by many independent persons as long ago as 1954. It might not be difficult to prove that “this was well-known to Babbage,” so no effort has been made to give credit to the originator, if indeed there was a unique originator.

Interlanguage Working

- Smooth interoperability between components written in different programming languages is a dream with a long history
- Distinct from, more ambitious and more interesting than, UNCOL
  - The benefits accrue to users, not to compiler-writers!
- Interoperability is more important than performance, especially for niche languages, e.g.
  - For years we thought nobody used functional languages because they were too slow
  - But a bigger problem was that you couldn't really write programs that did useful things (graphics, guis, databases, sound, networking, crypto,...)
  - We didn't notice, because we never tried to write programs which did useful things...
  - However, with languages like F# and Scale, interoperating via .Net resp. the JVM, things are changing …

Interlanguage Working

- Bilateral or Multilateral?
- Unidirectional or bidirectional?
- How much can be mapped?
- Explicit or implicit or no marshalling?
- What happens to the languages?
  - All within the existing framework?
  - Extended?
  - Pragmas or comments or conventions?
- External tools required?
- Work required on both sides of an interface?

Calling C bilaterally

- All compilers for high-level languages have some way of calling C
  - Often just hard-wired primitives for implementing libraries
    - Extensibility by recompiling the runtime system ☹
  - Sometimes a more structured FFI
  - Typically implementation-specific

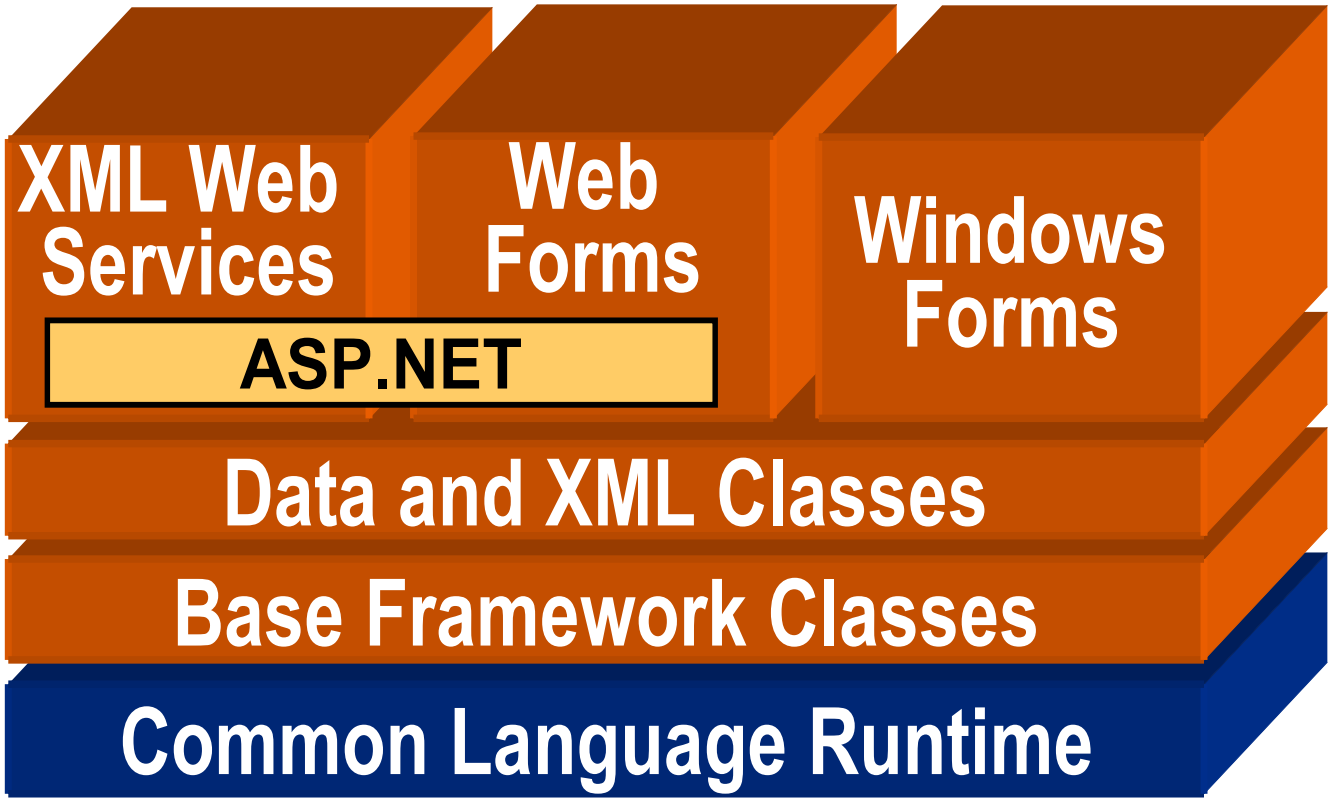
- Issues:
  - Data representation (31/32 bit ints, strings, record layout,...)
  - Calling conventions (registers, stack,..)
  - Storage management (especially copying collectors)

- It's a dirty job, but somebody's got to do it

Is there a better way?

- Well we saw the JVM earlier …
  - Most JVM support JNI
  - But this only works for calling from Java to C
  - Note HVM supports calling Java from C!
- And there are problems with languages which are not ”Java”-like
- What then? …

Common Programming Model - .NET



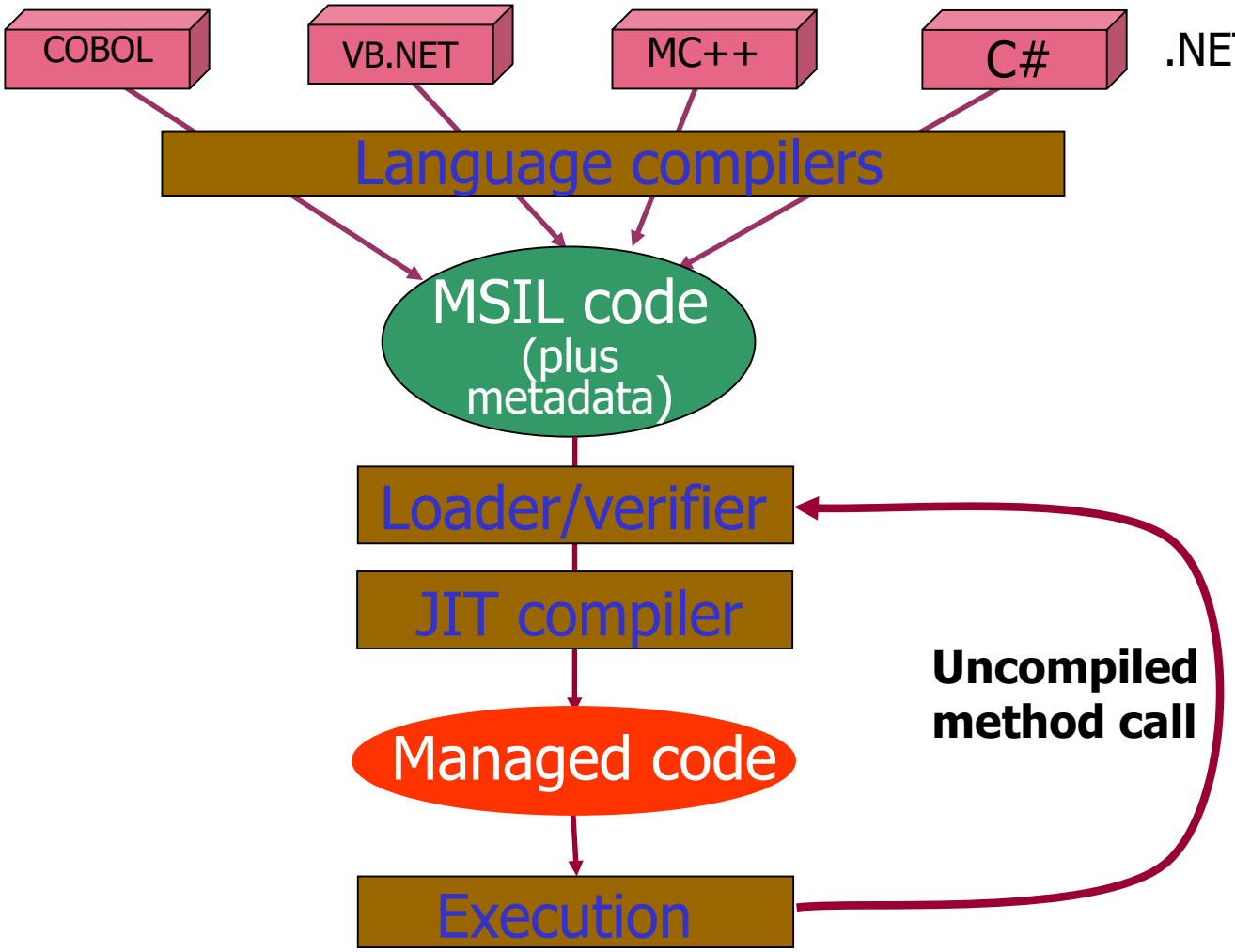
Overview of the CLI

- A common type system…
  …and a specification for language integration (**CLS**)
  - Execution engine with garbage collector and exception handling
  - Integral security system with verification
- A factored class library
  - A "modern" equivalent to the C runtime
- An intermediate language
  - CIL: Common Intermediate Language
- A file format
  - PE/COFF format, with extensions
  - An extensible metadata system
- Access to the underlying platform!

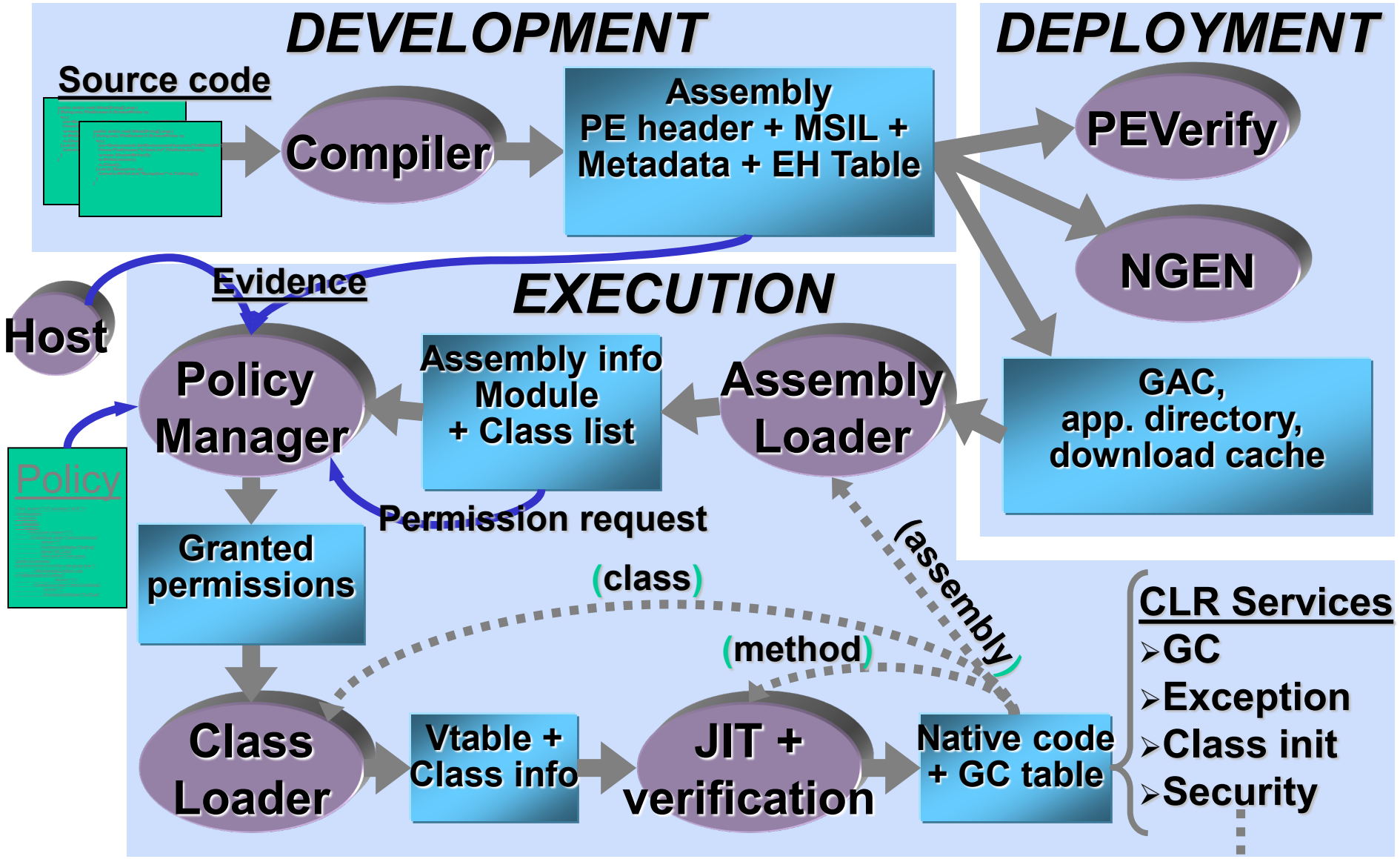
Terms to swallow

- CLI (Common Language Infrastructure)
- CLS (Common Language Specification)
- CTS (Common Type System)
- MSIL (Microsoft Intermediate Language)
  - CIL (Common Intermediate Language)
- CLR (Common Language Runtime)
- GAC (Global Assembly Cache)

Execution model



Managed Code Execution



What is the Common Language Runtime (CLR)?

- The CLR is the execution engine for .NET
- Responsible for key services:
  - Just-in-time compilation
  - heap management
  - garbage collection
  - exception handling
- Rich support for component software
- Language-neutral

The CLR Virtual Machine

- Stack-based, no registers
  - All operations produce/consume stack elements
  - Locals, incoming parameters live on stack
  - Stack is of arbitrary size; stack elements are “slots”
  - May or may not use real stack once JITted
- Core components
  - Instruction pointer (IP)
  - Evaluation stack
  - Array of local variables
  - Array of arguments
  - Method handle information
  - Local memory pool
  - Return state handle
  - Security descriptor

- Execution example

```
int add(int a, int b)
{
    int c = a + b;
    return c;
}
```

Offset	Instruction	Parameters
IL_0000	ldarg	0
IL_0001	ldarg	1
IL_0002	add	
IL_0003	stloc	0
IL_0004	ldloc	0
IL_0005	ret	

CIL Basics

- Data types
  - `void`
  - `bool`
  - `char`, `string`
  - `float32`, `float64`
  - [`unsigned`] `int8`, `int16`, `int32`, `int64`
  - `native` [`unsigned`] `int`: native-sized integer value
  - `object`: System.Object reference
  - Managed pointers, unmanaged pointers, method pointers(!)
- Names
  - All names must be assembly-qualified fully-resolved names
    - [*assembly*]*namespace*`.`*class*`::`*Method*
    - `[mscorlib]System.Object::WriteLine`

CIL Instructions

- Stack manipulation
  - `dup`: Duplicate top element of stack (pop, push, push)
  - `pop`: Remove top element of stack
  - `ldloc`, `ldloc.`*n*, `ldloc.s` *n*: Push local variable
  - `ldarg`, `ldarg.`*n*, `ldarg.s` *n*: Push method arg
    - "this" pointer arg 0 for instance methods
  - `ldfld` *type class::fieldname*: Push instance field
    - requires "this" pointer on top stack slot
  - `ldsfld` *type class::fieldname*: Push static field
  - `ldstr` *string*: Push constant string
  - `ldc.`<*type*> *n*, `ldc.`<*type*>`.`*n*: Push constant numeric
    - <type> is i4, i8, r4, r8

CIL Instructions

- Branching, control flow
  - `beq`, `bge`, `bgt`, `ble`, `blt`, `bne`, `br`, `brtrue`, `brfalse`
    - Branch target is label within code
  - `jmp` <*method*>: Immediate jump to method (goto, sort of)
  - `switch` (*t1, t2, ... tn*): Table switch on value
  - `call` *retval Class::method(Type, ...)*: Call method
    - Assumes arguments on stack match method expectations
    - Instance methods require "this" on top
    - Arguments pushed in right-to-left order
  - `calli` *callsite-description*: Call method through pointer
  - `ret`: Return from method call
    - Return value top element on stack

CIL Instructions

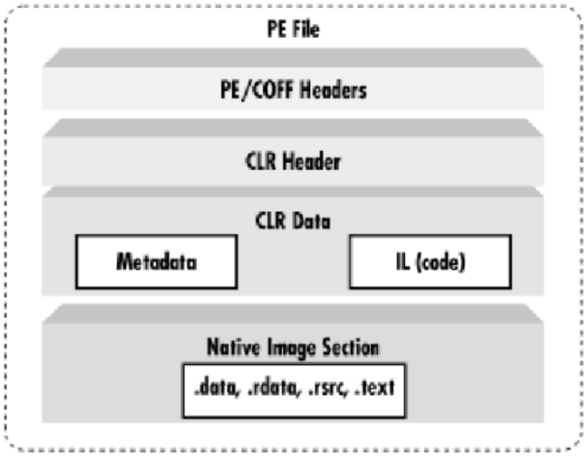
- Object model instructions
  - `newobj` *ctor*: Create instance using ctor method
  - `initobj` *type*: Create value type instance
  - `newarr` *type*: Create vector (zero-based, 1-dim array)
  - `ldelem`, `stelem`: Access vector elements
  - `isinst` *class*: Test cast (C# "is")
  - `castclass` *class*: Cast to type
  - `callvirt` *signature*: Call virtual method
    - Assumes "this" in slot 0--cannot be null
    - vtable lookup on object on *signature*
  - `box`, `unbox`: Convert value type to/from object instance

CIL Instructions

- Exception handling
  - `.try`: Defines guarded block
  - Dealing with exception
    - catch: Catch exception of specified type
    - fault: Handle exceptions but not normal exit
    - filter: Handle exception if filter succeeds
    - finally: Handle exception and normal exit
  - throw, rethrow: Put exception object into exception flow
  - leave: Exit guarded block

CIL assembler

- ILAsm (IL Assembly) closest to raw CIL
  - Assembly language
    - CIL opcodes and operands
    - Assembler directives
    - Intimately aware of the CLI (objects, interfaces, etc)
  - ilasm.exe (like JASMIN for Java/JVM)
  - Ships with FrameworkSDK, Rotor, along with a few samples
  - Creates a PE (portable executable) file (.exe or .dll)



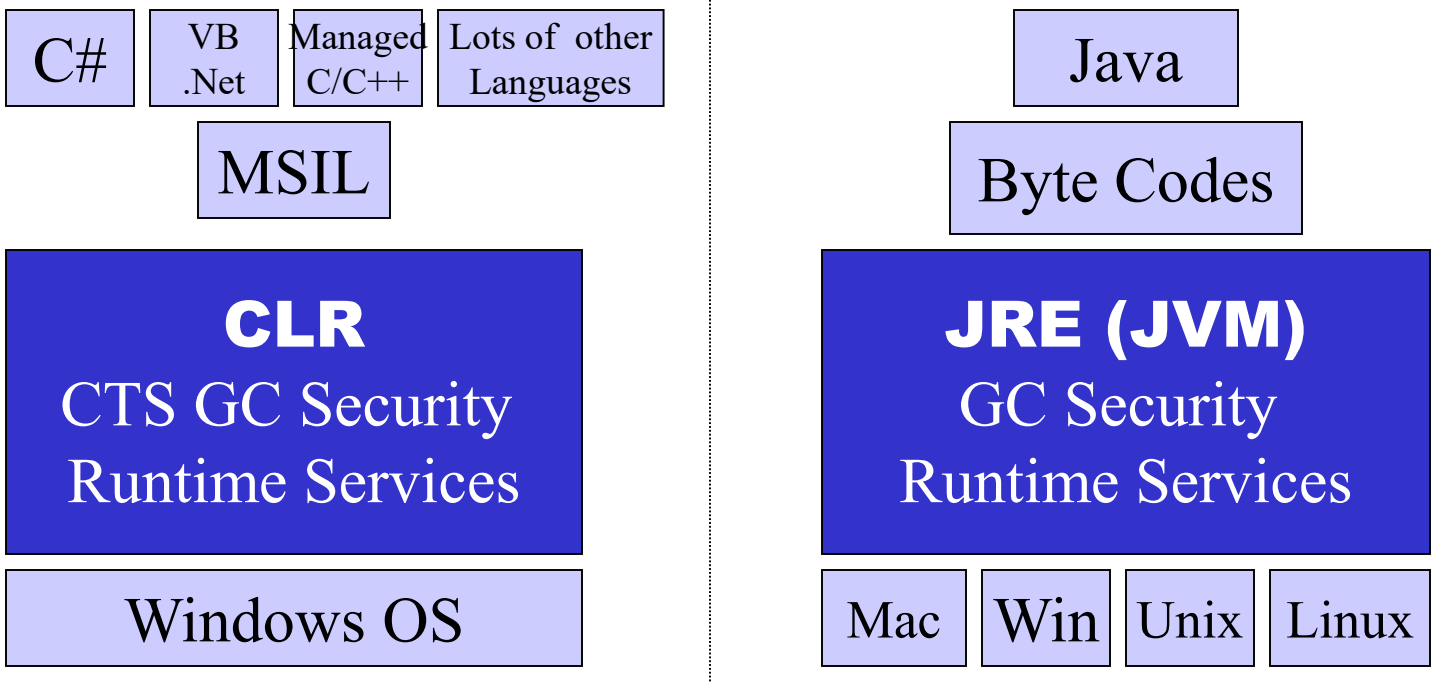
Example 1

- Hello, CIL!

```
.assembly extern mscorlib { }
.assembly Hello { }

.class private auto ansi beforefieldinit App
       extends [mscorlib]System.Object
{
  .method private hidebysig static void Main() cil managed
  {
    .entrypoint
    .maxstack  1
    ldstr      "Hello, CIL!"
    call       void [mscorlib]System.Console::WriteLine(string)
    ret
  } // end of method App::Main
} // end of class App
```

CLR vs JVM



Both are ‘middle layers’ between an intermediate language & the underlying OS

Java Byte Code and MSIL

- Java byte code (or JVML) is the low-level language of the JVM.
- MSIL (or CIL or IL) is the low-level language of the .NET Common Language Runtime (CLR).
- Superficially, the two languages look very similar.

JVML:

```
iload 1
iload 2
iadd
istore 3
```

MSIL:

```
ldloc.1
ldloc.2
add
stloc.3
```

- One difference is that MSIL is designed only for JIT compilation.
- The generic add instruction would require an interpreter to track the data type of the top of stack element, which would be prohibitively expensive.

JVM vs. CLR

- JVM's storage locations are all 32-bit therefore e.g. a 64-bit int takes up two storage locations
- The CLR VM allows storage locations of different sizes

- In the JVM all pointers are put into one reference type
- CLR has several reference types e.g. valuetype reference and reference type

JVM vs. CLR

- CLR provides ”typeless” arithmetic instructions
- JVM has separate arithmetic instruction for each type (iadd, fadd, imul, fmul...)

- JVM requires manual overflow detection
- CLR allows user to be notified when overflows occur

- Java has a maximum of 64K branches (if...else)
- No limit of branches in CLR

JVM vs. CLR

- JVM distinguishes between invoking methods and interface (invokevirtual and invokeinterface)
- CLR makes no distinction

- CLR supports tail calls (iteration in Scheme)
- Must resort to tricks in order to make JVM discard stack frames

Alternatives to JVM and CLR

- C
  - (or C++ or Java or C# or ..)
- JavaScript
- WebAssembly
- GENERIC, GIMPLE and RTL for gcc
- Dalvik VM
- LLVM IR

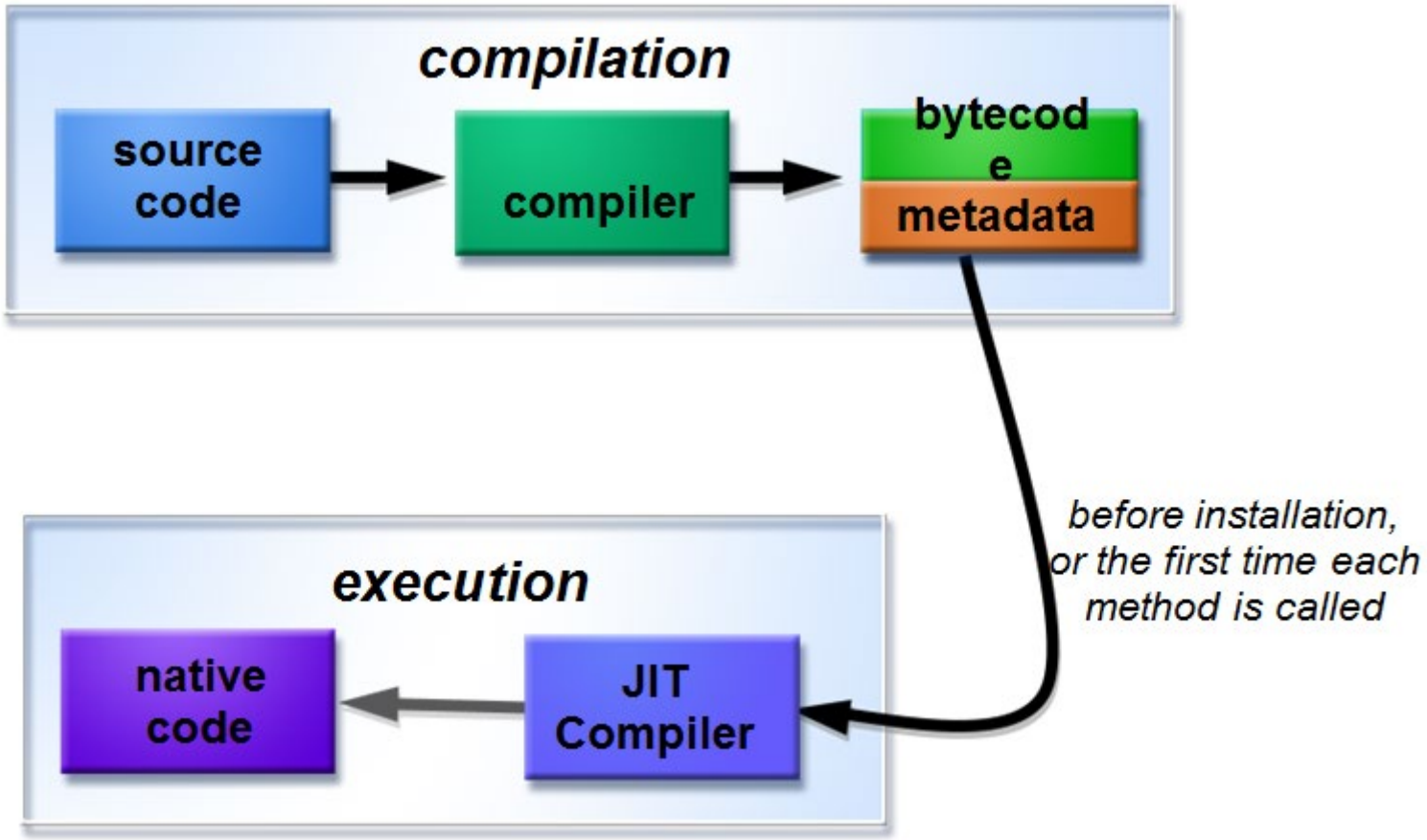
Comparison of Various VMs

Virtual machine	Machine model	Memory management	Code security	Interpreter	JIT	AOT	Shared libraries	Common Language Object Model	Dynamic typing
Android Runtime (ART)	register	automatic	Yes	No	No	Yes	?	No	No
BEAM (Erlang)	register	automatic	?	Yes	Yes	Yes	Yes	Yes	Yes
Common Language Runtime (CLR)	stack	automatic or manual	Yes	No	Yes	Yes	Yes	Yes	Yes
Dalvik	register	automatic	Yes	Yes	Yes	No	?	No	No
Dis (Inferno)	register	automatic	Yes	Yes	Yes	Yes	Yes	No	No
DotGNU Portable.NET	stack	automatic or manual	No	No	Yes	Yes	Yes	Yes	No
Java virtual machine (JVM)	stack	automatic	Yes	Yes	Yes	Yes	Yes	Yes	Yes[1]
JikesRVM	stack	automatic	No	No	Yes	No	?	No	No
LLVM	register	manual	No	Yes	Yes	Yes	Yes	Yes	No
Mono	stack	automatic or manual	Yes	Yes	Yes	Yes	Yes	Yes	Yes
Parrot	register	automatic	No	Yes	No[2]	Yes	Yes	Yes	Yes
Squeak	stack	automatic	No	Yes	Yes	No	Yes	No	Yes

http://en.wikipedia.org/wiki/Comparison_of_application_virtual_machines

Just-In-Time Compilation

- JIT compilers in JRE (JVM) and .NET runtimes



Just-In-Time Compilation (cont)

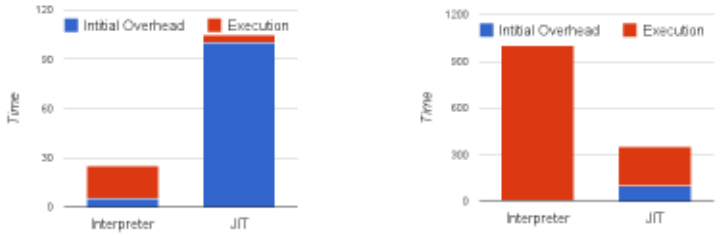
- At the time of code execution, the JIT compiler will compile some or all of it to native machine code for better performance.
  - Can be done per-file, per-function or even on any arbitrary code fragment (e.g. tracing JIT)
- The compiled code is cached and reused later without needing to be recompiled (unlike interpretation).

Java Virtual Machine - HotSpot

- Interpreter mode (-Xint)
- server mode (-server)
  - aggressive and complex optimizations
  - slow startup
  - fast execution
- client mode (-client)
  - less optimizations
  - fast startup
  - slower execution

Just-In-Time Overhead

JIT: 4x speedup, but 20x initial overhead



Matthew Arnold, Stephen Fink, David Grove, and Michael Hind, ACACES'06, 2006

What can you do with this in your project ?

- Consider generation code for a VM
  - JVM via JASMIN
  - CLR via ILASM
  - Some other VM
    - Python VM
    - Smalltalk VM
    - BEAM (Erlang)