Case Study: JVM

Virtual Machines

What is a machine?

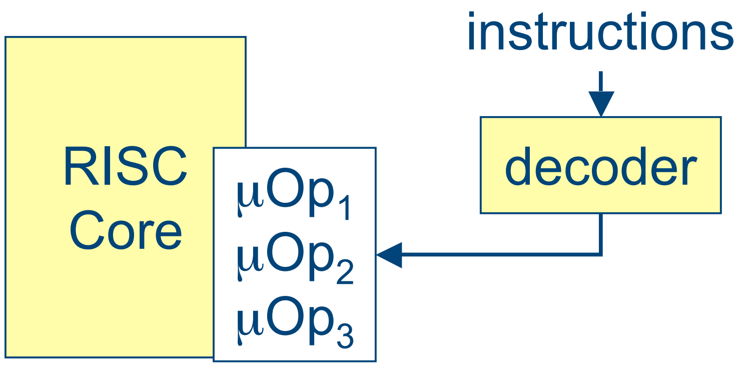
- does something (...useful)
- programmable
- concrete (hardware)

What is a virtual machine?

- a machine that is not concrete
- a software emulation of a physical computing environment

Reality is somewhat fuzzy!

Is a Pentium-II a machine?



Hardware and software are logically equivalent

(A. Tanenbaum)

Virtual Machine, Intermediate Language

- Pascal P-Code (1975)
  - stack-based processor
  - strong type machine language
  - compiler: one front end, many back ends
  - UCSD Apple][implementation, PDP 11, Z80
- Modula M-Code (1980)
  - high code density
  - Lilith as microprogrammed virtual processor
- JVM – Java Virtual Machine (1995)
  - Write Once – Run Everywhere
  - interpreters, JIT compilers, Hot Spot Compiler
- Microsoft .NET (2000)
  - language interoperability

JVM Case Study

- compiler (Java to bytecode)
- interpreter, ahead-of-time compiler, JIT
- dynamic loading and linking
- exception Handling
- memory management, garbage collection
- OO model with single inheritance and interfaces
- system classes to provide OS-like implementation
  - compiler
  - class loader
  - runtime
  - system

JVM: Type System

- Primitive types
  - byte
  - short
  - int
  - long
  - float
  - double
  - char

  - reference
  - boolean mapped to int
- Object types
  - classes
  - interfaces
  - arrays
- Single class inheritance
- Multiple interface implementation
- Arrays
  - anonymous types
  - subclasses of java.lang.Object

JVM: Java Byte-Code

Memory access

- tload / tstore
- ttload / ttstore
- tconst
- getfield / putfield
- getstatic / putstatic

Operations

- tadd / tsub / tmul / tdiv
- tshifts

Conversions

- f2i / i2f / i2l / ....
- dup / dup2 / dup_x1 / ...

Control

- ifeq / ifne / iflt / ....
- if_icmpeq / if_acmpeq
- invokestatic
- invokevirtual
- invokeinterface
- athrow
- treturn

Allocation

- new / newarray

Casting

- checkcast / instanceof

JVM: Java Byte-Code Example

bipush

Operation Push `byte`

Format

bipush
byte

Forms *bipush* = 16 (0x10)

Operand Stack ... => ..., *value*

Description The immediate *byte* is sign-extended to an `int` *value*. That *value* is pushed onto the operand stack.

JVM: Machine Organization

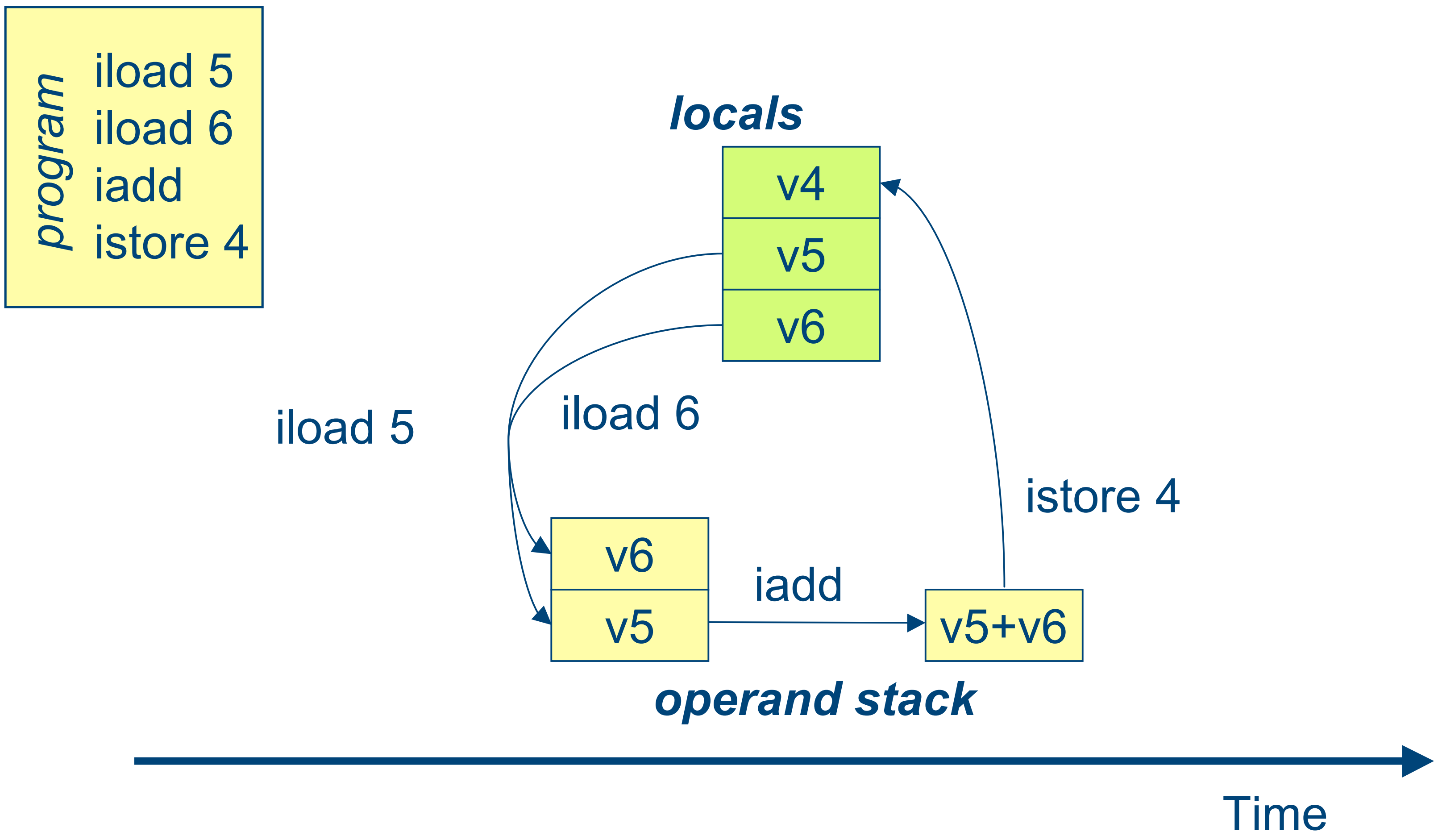
Virtual Processor

- stack machine
- no registers
- typed instructions
- no memory addresses, only symbolic names

Runtime Data Areas

- pc register
- stack
  - locals
  - parameters
  - return values
- heap
- method area
  - code
- runtime constant pool
- native method stack

JVM: Execution Example



JVM: Reflection

Load and manipulate *unknown* classes at runtime.

- java.lang.Class
  - getFields
  - getMethods
  - getConstructors
- java.lang.reflect.Field
  - setObject getObject
  - setInt getInt
  - setFloat getFloat
  - .....
- java.lang.reflect.Method
  - getModifiers
  - invoke
- java.lang.reflectConstructor

JVM: Reflection – Example

```
import java.lang.reflect.*;

public class ReflectionExample {

  public static void main(String args[]) {
    try {
      Class c = Class.forName(args[0]);
      Method m[] = c.getDeclaredMethods();
      for (int i = 0; i < m.length; i++) {
        System.out.println(m[i].toString());
      }
    } catch (Throwable e) {
      System.err.println(e);
    }
  }
}
```

JVM: Java Weaknesses

Transitive closure of java.lang.Object contains

- 1.1 47
- 1.2 178
- 1.3 180
- 1.4 248
- 5 (1.5) 280
- classpath 0.03 299

```
class Object
    public String toString();
    ..
}
```

```
class String {
    public String toUpperCase(Locale loc);
    ..
}
```

```
public final class Locale implements
        Serializable, Cloneable {
    ....
}
```

JVM: Java Weaknesses

Class static initialization

- T is a class and an instance of T is created

```
T tmp = new T();
```

- T is a class and a static method of T is invoked

```
T.staticMethod();
```

- A nonconstant static field of T is used or assigned
(field is not static, not final, and not initialized with compile-time constant)

```
T.someField = 42;
```

Problem

- circular dependencies in static initialization code

A

```
static {
    x = B.f();
}
```

B

```
static {
    y = A.f();
}
```

JVM: Java Weaknesses

```
interface Example {
  final static String labels[] = {“A”, “B”, “C”}
}
```

hidden static initializer:

```
  labels = new String[3];
  labels[0] = “A”; labels[1] = “B”; labels[2] = “C”;
```

Warning:

- in Java `final` means write-once!
- interfaces may contain code

JVM: Memory Model

- The JVM specs define a **memory model**:
  - defines the relationship between variables and the underlying memory
  - meant to guarantee the same behavior on every JVM
- The compiler is allowed to **reorder** operation unless **`synchronized`** or **`volatile`** is specified.

JVM: Reordering

- read and writes to ordinary variables can be reordered.

```
public class Reordering {
  int x = 0, y = 0;

  public void writer() {
    x = 1;
    y = 2;
  }

  public void reader() {
    int r1 = y;
    int r2 = x;
  }
}
```

JVM: Memory Model

- synchronized: in addition to specify a monitor it defines a **memory barrier**:
  - acquiring the lock implies an invalidation of the caches
  - releasing the lock implies a write back of the caches
- synchronized blocks on the same object are **ordered**.
- order among accesses to volatile variables is guaranteed (but **not** among volatile and other variables).

JVM: Double Checked Lock

Singleton

```
public class SomeClass {

  private Resource resource = null;

  public Resource synchronized getResource() {


        if (resource == null) {
          resource = new Resource();
        }


     return resource;
  }
}
```

JVM: Double Checked Lock

Double checked locking

```
public class SomeClass {

  private Resource resource = null;

  public Resource              getResource() {
    if (resource == null) {
      synchronized {
        if (resource == null) {
          resource = new Resource();
        }
      }
    }
    return resource;
  }
}
```

JVM: Double Checked Lock

Thread 1

```
public class SomeClass {

  private Resource resource
    = null;

   public Resource getResource() {
     if (resource == null) {
       synchronized {
         if (resource == null) {
           resource =
             new Resource();
         }
       }
     }
     return resource;
  }
}
```

Thread 2

```
public class SomeClass {

  private Resource resource
    = null;

   public Resource getResource() {
     if (resource == null) {
       synchronized {
         if (resource == null) {
           resource =
             new Resource();
         }
       }
     }
     return resource;
```

The object is
instantiated
but not yet initialized!

JVM: Immutable Objects are not Immutable

- Immutable objects:
  - all types are primitives or references to immutable objects
  - all fieds are final
- Example (simplified): java.lang.String
  - contains
    - an array of characters
    - the length
    - an offset
  - example: s = “abcd”, length = 2, offset = 2, string = “cd”

```
String s1 = "/usr/tmp"
String s2 = s1.substring(4); //should contain "/tmp"
```

- Sequence: s2 is instantiated, the fields are initialized (to 0), the array is copied, the fields are written by the constructor.
- What happens if instructions are reordered?

JVM: Reordering Volatile and Nonvolatile Stores

- volatile reads and writes are totally ordered among threads
- but not among normal variables
- example

```
volatile boolean initialized = false;
SomeObject o = null;
```

Thread 1

```
o = new SomeObject;
initialized = true;
```

?

Thread 2

```
while (!initialized) {
  sleep();
}
o.field = 42;
```


JVM: JSR 133

- Java Community Process
- Java memory model revision

- Final means final
- Volatile fields cannot be reordered

Java JVM: Execution

- Interpreted (e.g., Sun JVM)
  - bytecode instructions are interpreted sequentially
  - the VM emulates the Java Virtual Machine
  - slower
  - quick startup
- Just-in-time compilers (e.g., Sun JVM, IBM JikesVM)
  - bytecode is compiled to native code at load time (or later)
  - code can be optimized (at compile time or later)
  - quicker
  - slow startup
- Ahead-of time compilers (e.g., GCJ)
  - bytecode is compiled to native code offline
  - quick startup
  - quick execution
  - static compilation

JVM: Loader – The Classfile Format

```
ClassFile {
    version
    constant pool
    flags
    super class
    interfaces
    fields
    methods
    attributes
}
```

Constants:

- Values
  String / Integer / Float / ...
- References
  Field / Method / Class / ...

Attributes:

- ConstantValue
- Code
- Exceptions

JVM: Class File Format

```
class HelloWorld {

  public static void printHello() {
      System.out.println("hello, world");
  }

  public static void main (String[] args) {
      HelloWorld myHello = new HelloWorld();
      myHello.printHello();
  }

}
```

JVM: Class File (Constant Pool)

1. String hello, world
2. Class HelloWorld
3. Class java/io/PrintStream
4. Class java/lang/Object
5. Class java/lang/System
6. Methodref HelloWorld.<init>()
7. Methodref java/lang/Object.<init>()
8. Fieldref java/io/PrintStream java/lang/System.out
9. Methodref HelloWorld.printHello()
10. Methodref java/io/PrintStream.println(java/lang/String)
11. NameAndType <init> ()V
12. NameAndType out Ljava/io/PrintStream;
13. NameAndType printHello ()V
14. NameAndType println (Ljava/lang/String;)V
15. Unicode ()V
16. Unicode (Ljava/lang/String;)V
17. Unicode ([Ljava/lang/String;)V
18. Unicode <init>
19. Unicode Code
20. Unicode ConstantValue
21. Unicode Exceptions
22. Unicode HelloWorld
23. Unicode HelloWorld.java
24. Unicode LineNumberTable
25. Unicode Ljava/io/PrintStream;
26. Unicode LocalVariables
27. Unicode SourceFile
28. Unicode hello, world
29. Unicode java/io/PrintStream
30. Unicode java/lang/Object
31. Unicode java/lang/System
32. Unicode main
33. Unicode out
34. Unicode printHello

JVM: Class File (Code)

Methods

```
0  <init>()
     0  ALOAD0
     1  INVOKESPECIAL  [7]   java/lang/Object.<init>()
     4  RETURN

1  PUBLIC STATIC main(java/lang/String  [])
     0  NEW  [2]   HelloWorld
     3  DUP
     4  INVOKESPECIAL  [6]   HelloWorld.<init>()
     7  ASTORE1
     8  INVOKESTATIC  [9]   HelloWorld.printHello()
    11  RETURN

2  PUBLIC STATIC printHello()
     0  GETSTATIC  [8]   java/io/PrintStream  java/lang/System.out
     3  LDC1   hello, world
     5  INVOKEVIRTUAL  [10]   java/io/PrintStream.println(java/lang/String  )
     8  RETURN
```

JVM: Compilation – Pattern Expansion

- Each byte code is translated according to fix patterns
  - \+ easy
  - \- limited knowledge
- Example (pseudocode)

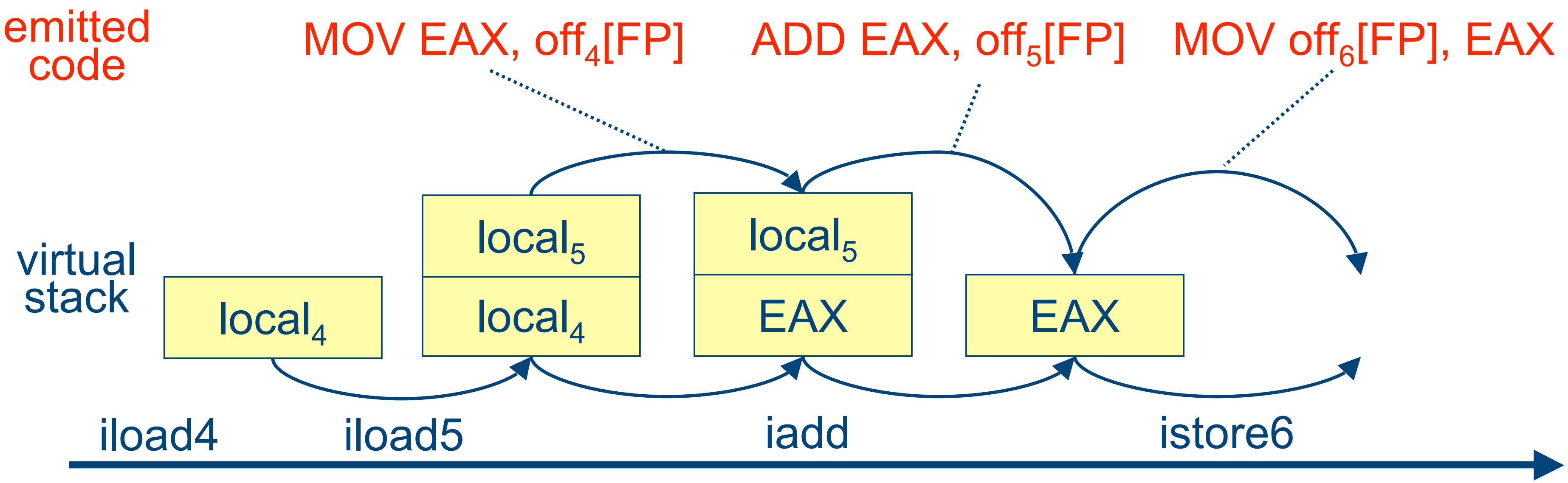
```
switch (o) {
case ICONST<n>: generate("push n"); PC++; break;
case ILOAD<n>: generate("push off_n[FP]"); PC++; break;
case IADD: generate("pop -> R1");
           generate("pop -> R2");
           generate("add R1, R2 -> R1");
           generate("push R1");
           PC++;
           break;
...
```

JVM: Optimizing Pattern Expansion

Main Idea:

- use internal **virtual stack**
- stack values are consts / fields / locals / array fields / registers / ...
- flush stack **as late as possible**

```
iload 4
iload 5
iadd
istore 6
```



JVM: Compiler Comparison

```
iload_4
iload_5
iadd
istore_6
```

5 instructions
9 memory accesses

3 instructions
3 memory accesses

pattern expansion

```
push off4[FP]
push off5[FP]
pop  EAX
add  0[SP], EAX
pop  off6[FP]
```

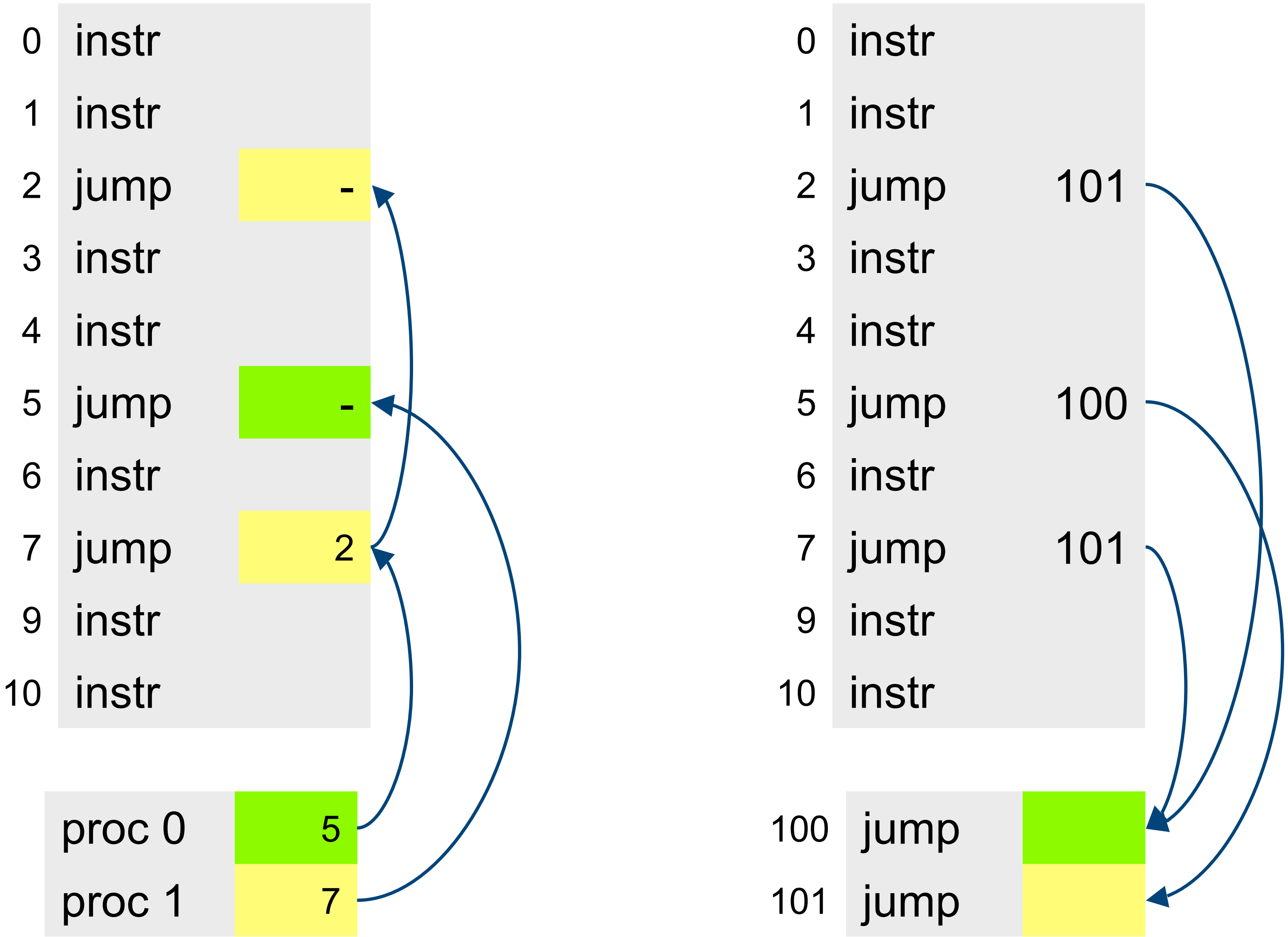
optimized

```
mov EAX, off4[FP]
add EAX, off5[FP]
mov off6[FP], EAX
```

Linking (General)

- A compiled program contains references to **external code** (libraries)
- After loading the code the system need to **link** the code to the library
  - identify the calls to external code
  - locate the callees (and load them if necessary)
  - patch the loaded code
- Two options:
  - the code contains a list of sites for each callee
  - the calls to external code are jumps to a *procedure linkage table* which is then patched (double indirection)

Linking (General)



Linking (General)

0	instr	
1	instr	
2	jump	&p1
3	instr	
4	instr	
5	jump	&p0
6	instr	
7	jump	&p1
9	instr	
10	instr	

proc 0	5
proc 1	7

0	instr	
1	instr	
2	jump	101
3	instr	
4	instr	
5	jump	100
6	instr	
7	jump	101
9	instr	
10	instr	

100	jump	&p0
101	jump	&p1

JVM: Linking

- Bytecode interpreter
  - references to other objects are made through the JVM (e.g., invokevirtual, getfield, …)
- Native code (ahead of time compiler)
  - static linking
  - classic native linking
- JIT compiler
  - only some classes are compiled
  - calls could reference classes that are not yet loaded or compiled (delayed compilation)

  ➔ code instrumentation

JVM: Methods and Fields Resolution

- method and fields are accessed through special VM functions (e.g., invokevirtual, getfield, …)
- the parameters of the special call defines the target
- the parameters are indexes in the constant pool
- the VM checks id the call is legal and if the target is presentl

JVM: JIT – Linking and Instrumentation

- Use code instrumentation to detect first access of static fields and methods

```
class A {

   ....
   ...B.x
}
```

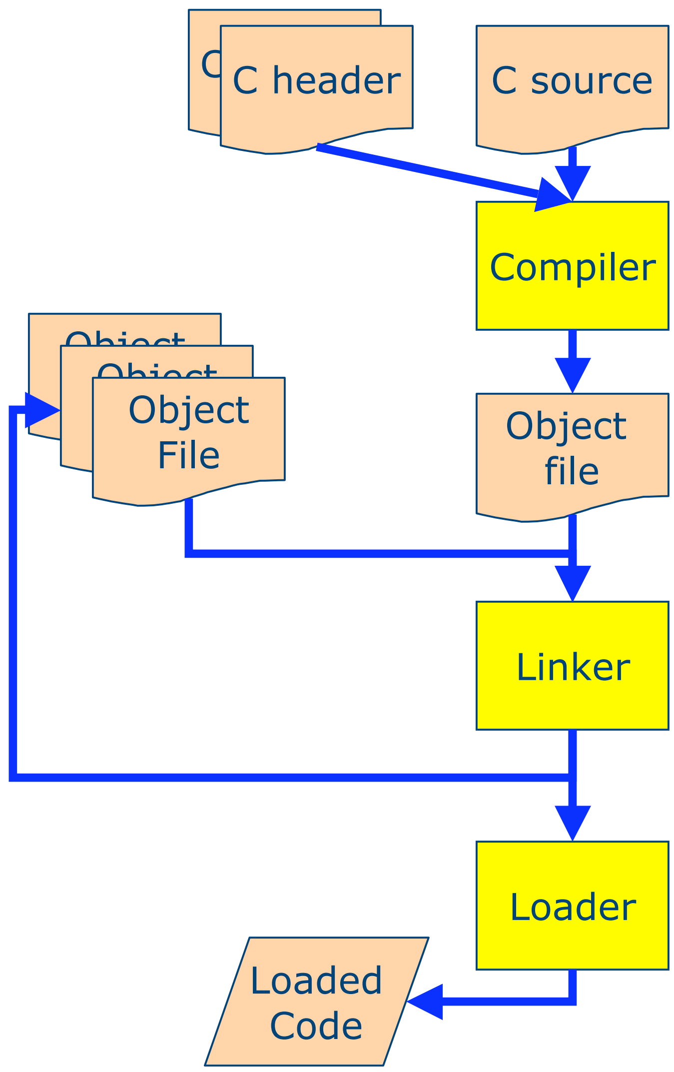
```
class B {
   int x;
}
```

```
B.x
```

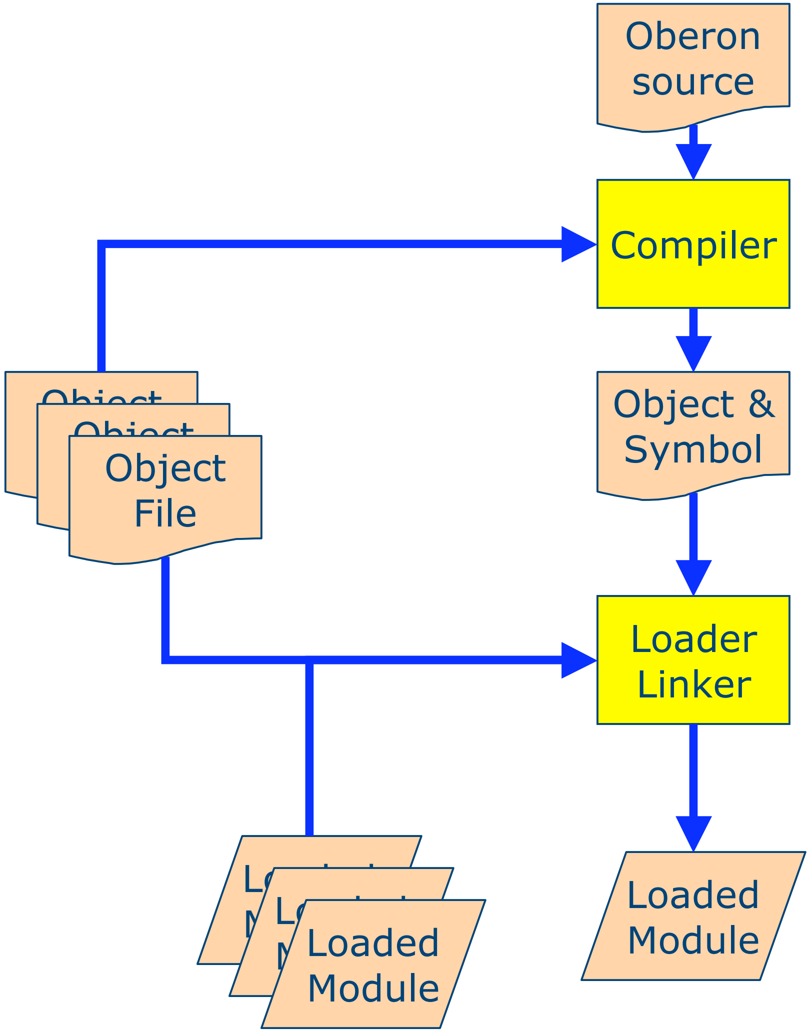
```
CheckClass(B);
B.x
```

```
IF ~B.initialized
THEN
   Initialize(B)
END;
```

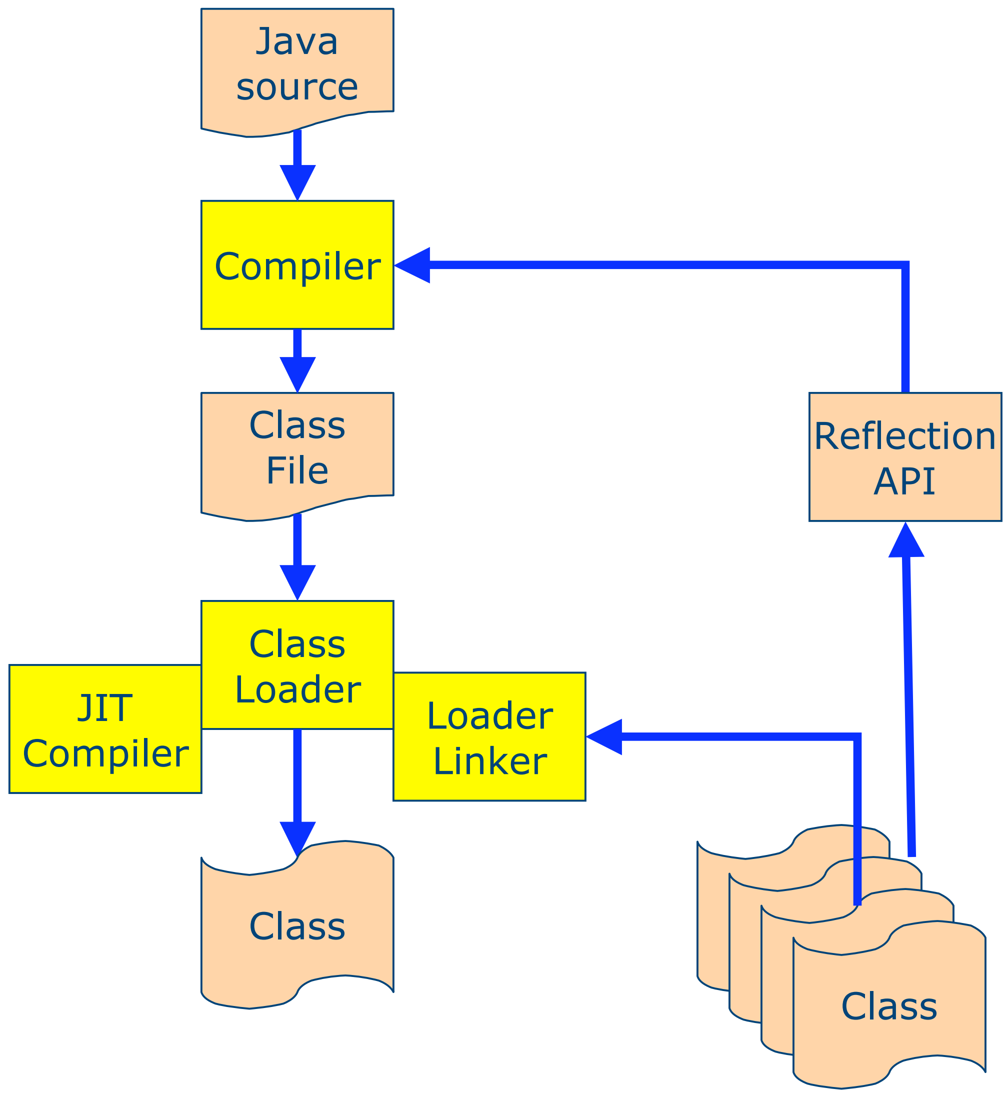

Compilation and Linking Overview



Compilation and Linking Overview



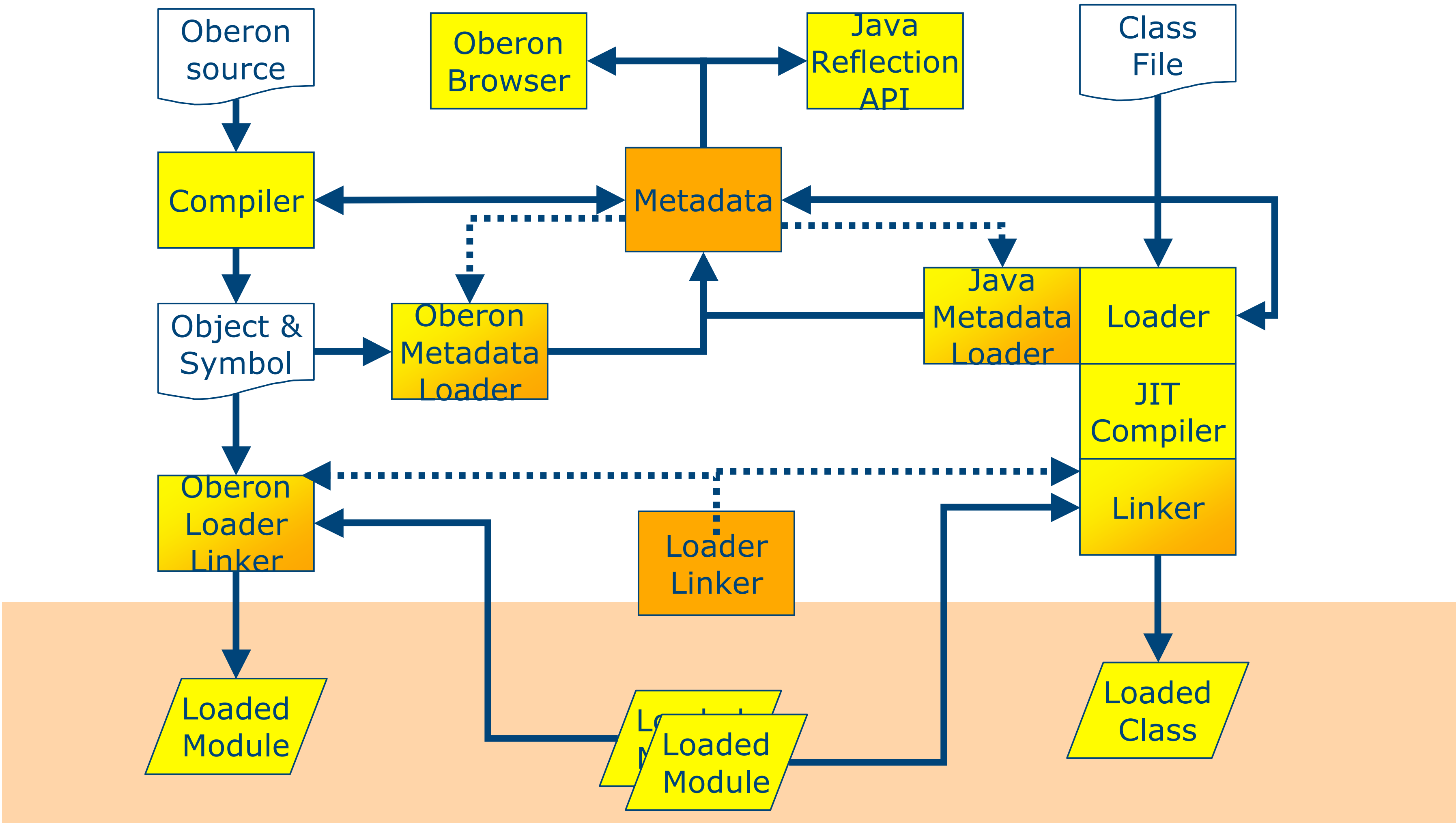
Compilation and Linking Overview



Jaos

- Jaos (**J**ava on **A**ctive **O**bject **S**ystem) is a Java virtual machine for the Bluebottle system
- goals:
  - implement a JVM for the Bluebottle system
  - show that the Bluebottle kernel is generic enough to support more than one system
  - interoperability between the Active Oberon and Java languages
  - interoperability between the Oberon System and the Java APIs

Jaos (Interoperability Framework)



JVM: Verification

- Compiler generates "good" code....
- .... that could be changed before reaching the JVM

➔ need for verification

Verification makes the VM simpler (less run-time checks):

- no operand stack overflow
- load / stores are valid
- VM types are correct
- no pointer forging
- no violation of access restrictions
- access objects as they are (type)
- local variable initialized before load
- …

JVM: Verification

Pass1 (Loading):

- class file version check
- class file format check
- class file complete

Pass 2 (Linking):

- final classes are not subclassed
- every class has a superclass (but Object)
- constant pool references
- constant pool names

JVM: Verification

Delayed for performance reasons

Pass 3 (Linking):

For each operation in code (independent of the path):

- operation stack size is the same
- accessed variable types are correct
- method parameters are appropriate
- field assignment with correct types
- opcode arguments are appropriate

Byte-Code Verification

Pass 4 (RunTime):

First time a type is referenced:

- load types when referenced
- check access visibility
- class initialization

First member access:

- member exists
- member type same as declared
- current method has right to access member

JVM: Byte-Code Verification

Verification:

- branch destination must exists
- opcodes must be legal
- access only existing locals
- code does not end in the middle of an instruction
- types in byte-code must be respected
- execution cannot fall of the end of the code
- exception handler begin and end are sound

JVM: Bootstrapping

How to start a JVM?

- External help needed!
- Load core classes
- Compile classes
- Provide memory management
- Provide threads

Solution:

Implement Java on 3rd party system

- Linux
- Solaris
- Windows
- Bluebottle
- Java

Bootstrapping: Jaos Example

- All native methods in Active Oberon
- Use Bluebottle run-time structures
  - module (= class)
  - type descriptor
  - object (= object instance)
  - active object (= thread)
- Bootstrap
  - load core classes
    - Object, String, System, Runtime, Threads, ...
    - Exception
  - forward exception to java code
  - allocate java classes from Oberon

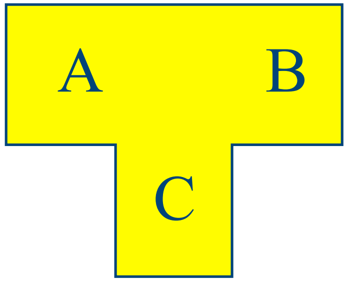
Bootstrapping: Jnode VM Example

- JVM written in Java
- small core in assembler
  - low-level functionalities that requires special assembler instructions
- some native methods inlined by the compiler
  - Unsafe
    - debug(String)
    - int AddressToInt(Address)
    - int getInt(Object, offset)
- Bootstrap
  - compile to Java classes
  - bootloader:
    - native compilation
    - code placement
    - structure allocation
    - make boot image
  - boot with GNU/GRUB

Bootstrapping: Oberon / Bluebottle

- Compile each module to machine code
  - system calls for
    - newrec (record)
    - newsys (block, no ptrs)
    - newarr (array)
  - linker / bootlinker patches syscalls with real procedures
- bootlinker is same as linker, but uses different memory addresses
  - simulates memory allocation
  - start address configurable
  - glue code to call all module bodies

Bootstrapping Compilers



This compiler is written in language “C” and translates programs in “A” into programs in “B”