Many High-Level Languages

Choosing a programming language...

▶ Basic
▶ Fortran 95
▶ Ada
▶ Python
▶ Haskell
▶ C++
▶ Java

We choose Java because it has fewer ways “to shoot yourself in the foot.”
Terminology

- compiling (originally): linking subroutines
- translating: converting from a high-level language to a low-level
- compiling: translating to native (of some real machine) code
- batch: compile once, run many times
- interpreting: running the program under the control of a software program
- interactive: read-eval-print loop. Evaluate means interpret in this case; but now it could be extended to compile and run.
Implementation

A language may be translated or implemented by zero, one, or many different systems. Unqualified statements such as

1. “Language X is compiled.”
2. “Language X is slow.”

are nonsense, because the speed of execution and the type of translation depend on the implementation. Of course, the language may influence or seek to influence the implementation. A language may be closely associated with a particular implementation. But a language itself is not an implementation.

The world of language implementations is often quite complex. For example, there are the GNU gfortran and g77 compilers, not to mention many commercial compilers for Fortran. Also, there are the SUN JDK tools for Java, the IBM Jikes compiler for Java, and the GNU gcj compiler.
In its usual meaning, a compiler is one that collects and edits material written by others into a collection.

... compiled by Carl Parrish, ... edited by F. Bauer and J. Eickel
A compiler was originally a program that “compiled” subroutines. When in 1954 the combination “algebraic compiler” came into use, or rather into misuse, the meaning of the term had already shifted into the present one.

Traditional Compilation

source program

compiler

machine code

input → execute → output
How do you solve a hard problem?
How do you solve a hard problem?

One important approach is to break it into well-defined subproblems. (A compiler is just a big program.)
Compilation Steps

When examined in more detail, compilation takes several steps.

1. preprocessing, macro processing
2. translation (compiling)
3. assembling mnemonics
4. linking other code and preparing for execution

Macros (an extremely dangerous facility) are common in C and C++. Java does preprocessing to translate character sets.
Language Systems

Language translation and execution systems are big and complex these days because computers can execute larger and larger programs faster and faster. The programmer or program user rarely sees the individual steps.

IDEs, interactive language systems, JIT compilers, incremental compilers, and dynamic linking all conspire to hide and blur the important individual steps. (But make programming development faster and easier).
Assembly

1. **Source Program**
2. **Compile**
   - **Precompiled Libraries**
   - **Other Relocatable Code**
   - **Relocatable Code**
   - **Runtime System**
     - Interface to OS, Garbage Collection, Exception Handling, etc.
3. **Link (and Load)**
4. **Executable Module**
5. **Input** → **Execute** → **Output**
Interpreting

An *interpreter* is a program that takes another program as input and executes it, possibly line-by-line, possibly without translating it to an intermediate form. Sometimes the translation is to an intermediate form which may be executed by a *virtual* or *abstract machine*. Examples of abstract machines include: Forth virtual machine, p-code machine (Pascal), Python virtual machine, SECD machine (lambda calculus), Smalltalk virtual machine, Warren Abstract machine (Prolog).

As hardware gets faster, the advantage of portability overtakes the disadvantage of slow emulation, and multi-language virtual machines are becoming more important: the Microsoft .Net platform (C#, F#, Managed C++, Python) and the Java virtual machine (Java, Jython, Ada, and many other languages). Since these abstract machines execute complex source languages the machines must also provide the runtime support these languages expect.
Interpreting (continued)

Since an abstract machine may be abstract by virtue of having abstract instructions or by having abstract capabilities, the term abstract/virtual machine may be ambiguous and lead to confusion. Abstract instructions are likely to be slower than real instructions because of the extra software overhead of interpretation. Abstract capabilities are likely to be faster than programmer-supplied code because of the skill of the implementors and the use of the underlying machine.

The key aspect of an interpreter is emulation. The key aspect of a runtime system is support of functionality.
Superficially, we equate *abstract* and *virtual* machine. Technically, *abstract* conotates emulation, and *virtual* functionality. Hence, JVM is so-called to emphasize that the computing base of Java is beyond a mere ordinary machine and it does not mean the language is emulated. The base could be realized in hardware (but attempts so-far have not proved popular). JVMs could be interpreters, JITs, or the native executable code from compilers.
Modern, high-level languages require that a program have additional support during execution. This is sometimes called the runtime system. The runtime system contains lots of code that is not written by the programmer, but was written by others and used when a program in the language is run.

The runtime system may provide support for mathematical operations (e.g., exponentiation), floating-point arithmetic, complex numbers, high-level input and output functions, concurrency, memory management (e.g., garbage collection), etc. Modern languages tend to have larger and larger support systems.

The work of the runtime system may require assistance of the translation system, for example, to insert reference counting code, debugging code, etc. The runtime system must be available to every program in the language so it can run correctly, but none of the functionality might actually be used.
The distinction between the runtime system and the standard libraries is not always clear. Take these two statements in Java:

```java
System.out.printf("%d %s", 4, this);
new Thread().start();
```

Both statements appear to be just simple calls to library routines, but ultimately considerable code gets executed which the programmer did not, could not, or would not write (in Java).
Back to translation ...
Important Unix Tools

- gcc
- gas
- gdb
- make
- objdump
- uname
- od
Compilation — gcc

```
#include <stdio.h>

int main () {
  fputs ("Hello world!\n", stdout);
  return 0;
}
```
%gcc -o hello -v hello.c
Reading specs from /software/solaris/gnu/lib/gcc-lib/sparc-sun-solaris2.6/2.95.3/specs
gcc version 2.95.3 20010315 (release)
  /software/solaris/gnu/lib/gcc-lib/sparc-sun-solaris2.6/2.95.3/cpp0
    -lang-c -v -D__GNUC__=2 -D__GNUC_MINOR__=95 -Dsparc -Dsun -Dunix -D__svr4__ -D__SVR4 -D__sparc__ -D__sun__
    -Asystem(unix) -Asystem(svr4) -D__GCC_NEW_VARARGS__ -Acpu(sparc) -Amachine(sparc) hello.c /var/tmp/cc5V4Wy1.i
GNU CPP version 2.95.3 20010315 (release) (sparc)
#include "..." search starts here:
#include <...> search starts here:
/software/solaris/gnu/include
/software/solaris/gnu/lib/gcc-lib/sparc-sun-solaris2.6/2.95.3/../../../../sparc-sun-solaris2.6/include
/software/solaris/gnu/lib/gcc-lib/sparc-sun-solaris2.6/2.95.3/include
/usr/include
End of search list.
The following default directories have been omitted from the search path:
/software/solaris/gnu/lib/gcc-lib/sparc-sun-solaris2.6/2.95.3/../../../../include/g++-3
End of omitted list.
/software/solaris/gnu/lib/gcc/lib/gcc-lib/sparc-sun-solaris2.6/2.95.3/cc1
  /var/tmp/cc5V4Wy1.i -quiet -dumpbase hello.c -version -o /var/tmp/cc47fQVU.s
GNU C version 2.95.3 20010315 (release) (sparc-sun-solaris2.6) compiled by GNU C version 3.0.3.
/software/solaris/gnu/bin/as -V -Qy -s -o /var/tmp/ccNHrBWS.o /var/tmp/cc47fQVU.s
GNU assembler version 2.11.2 (sparc-sun-solaris2.6) using BFD version 2.11.2
/software/solaris/gnu/lib/gcc/lib/gcc-lib/sparc-sun-solaris2.6/2.95.3/collect2
  -V -Y P, /usr/ccs/lib:/usr/lib -Qy -o hello /software/solaris/gnu/lib/gcc-lib/sparc-sun-solaris2.6/2.95.3/cc1
GNU ld version 2.11.2 (with BFD 2.11.2)
  Supported emulations:
  elf32_sparc
cs> gcc -S hello.c -o hello.s
cs> gcc -S hello.c -o hello.s

.globl main
.type main, #function
.proc 04
main:

!#PROLOGUE# 0
.save %sp, -112, %sp
!#PROLOGUE# 1
.sethi %hi(.LLC0), %o1
.or %o1, %lo(.LLC0), %o0
.sethi %hi(__iob+16), %o2
.or %o2, %lo(__iob+16), %o1
.call puts, 0
.nop
.mov 0, %i0
.b .LL2
.nop

gcc compiles C to native code
Compilation

hello.o: file format elf32-sparc

Contents of section .text:
  0000 9de3bf90 13000000 90126000 15000000 ........`.....
  0010 9212a000 40000000 01000000 b0102000 ....@.........
  0020 10800002 01000000 81c7e008 81e80000 .............

Contents of section .data:

Contents of section .rodata:

Contents of section .comment:

Disassembly of section .text:

00000000 <main>:
  0: 9d e3 bf 90 save %sp, -112, %sp
  4: 13 00 00 00 sethi %hi(0), %o1
  8: 90 12 60 00 mov %o1, %o0 ! 0 <main>
  c: 15 00 00 00 sethi %hi(0), %o2
  10: 92 12 a0 00 mov %o2, %o1 ! 0 <main>
  14: 40 00 00 00 call 14 <main+0x14>
  18: 01 00 00 00 nop
  1c: b0 10 20 00 clr %i0 ! 0 <main>
  20: 10 80 00 02 b 28 <main+0x28>
  24: 01 00 00 00 nop
  28: 81 c7 e0 08 ret
  2c: 81 e8 00 00 restore
ELF – Executable and Linkable Format

**Linking View**

- ELF header
- Program header table (optional)
- section 1
- ...
- section n
- ...
- ...
- Section header table

**Execution View**

- ELF header
- Program header table
- Segment 1
- Segment 2
- ...
- ...
- Section header table (optional)
**ELF – Executable and Linkable Format**

typedef struct {
  unsigned char e_ident[16]; /* version and other info */
  uint16_t e_type; /* none, relocatable, executable, shared, core */
  uint16_t e_machine; /* none, SPARC, Intel, Motorola, MIPS, ... */
  uint32_t e_version;
  uintN_t e_entry; /* entry point */
  ...;
} ElfN_Ehdr;

**Note** `readelf (Unix)` and `elfdump (Solaris)` view elf files. **Note** `otool (Darwin)` to view Mach-o files.
Mach-O
Mach-O

(Pronounced “macho.”)

/* From # include <mach-o/loader.h> */
/* Mach header of the object file for 32-bit architectures. */
struct mach_header {
    uint32_t magic; /* mach magic number identifier */
    cpu_type_t cputype; /* PowerPC, I386 */
    cpu_subtype_t cpusubtype; /* machine specifier */
    uint32_t filetype; /* object, executable, shared, core, ....*/
    uint32_t ncmds; /* number of load commands */
    uint32_t sizeofcmds; /* the size of all the load commands */
    uint32_t flags; /* flags */
};

/* Constant for the magic field of the mach_header (32-bit architectures) */
#define MH_MAGIC 0xfeedface /* the mach magic number */
#define MH_CIGAM 0xcefaedfe /* NXSwapInt(MH_MAGIC) */
A wide range of techniques are used in translating Java into executable form. Several translators exist for the language.

1. IBM jikes
2. GNU gcj
3. Sun Java 2 SDK
Translating Java

A wide range of techniques are used in translating Java into executable form. Several translators exist for the language.

1. IBM jikes
2. GNU gcj
3. Sun Java 2 SDK

We begin by looking at GNU gcj to see a traditional translator in action. Then we move to the SUN Java 2 SDK and see the important role of byte code.
public class Hello {

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

}
Compilation — gcj

Reading specs from /software/solaris/gnu/lib/gcc-lib/sparc-sun-solaris2.9/3.3.2/specs
Reading specs from /software/solaris/gnu/lib/gcc-lib/sparc-sun-solaris2.9/3.3.2/../../../libgcj.spec
rename spec lib to liborig
Configured with: ./configure --prefix=/software/solaris/gnu --with-ld=/software/solaris/gnu/bin/ls --with-as=/software/solaris/gnu/as --enable-threads=posix --with-local-prefix=/software/solaris/cmn
Thread model: posix
gcc version 3.3.2
/software/solaris/gnu/lib/gcc-lib/sparc-sun-solaris2.9/3.3.2/jc1 Hello.java -fuse-divide-subroutine -fcheck-references -fuse-boehm-gc -fkeep-inline-functions -quiet -dumpbase Hello.java -auxbase Hello -g1 -version -o /var/tmp//ccgEgJBv.s
GNU Java version 3.3.2 (sparc-sun-solaris2.9)
compiled by GNU C version 2.95.3 20010315 (release).
GGC heuristics: --param ggc-min-expand=47 --param ggc-min-heapsize=32768
Class path starts here:
  ./
   /software/solaris/gnu/share/java/libgcj-3.3.2.jar/ (system) (zip)
   /software/solaris/gnu/lib/gcc-lib/sparc-sun-solaris2.9/3.3.2/../../../sparc-sun-solaris2.9/bin/as -V -Qy -
GNU assembler version 2.14 (sparc-sun-solaris2.9) using BFD version 2.14 20030612
/software/solaris/gnu/lib/gcc-lib/sparc-sun-solaris2.9/3.3.2/jvgenmain Hellomain /var/tmp//ccWJ2hCQ.i
/software/solaris/gnu/lib/gcc-lib/sparc-sun-solaris2.9/3.3.2/ccl /var/tmp//ccWJ2hCQ.i -quiet -dumpbase Hellomain
GNU C version 3.3.2 (sparc-sun-solaris2.9)
compiled by GNU C version 2.95.3 20010315 (release).
GGC heuristics: --param ggc-min-expand=47 --param ggc-min-heapsize=32768
/software/solaris/gnu/lib/gcc-lib/sparc-sun-solaris2.9/3.3.2/../../../sparc-sun-solaris2.9/bin/as -V -Qy -
GNU assembler version 2.14 (sparc-sun-solaris2.9) using BFD version 2.14 20030612
/software/solaris/gnu/lib/gcc-lib/sparc-sun-solaris2.9/3.3.2/collect2 -V -Y P,/usr/ccs/lib:/usr/lib -Qy -o hel...
GNU ld version 2.14 20030612
Supported emulations:
elf32_sparc
elf64_sparc
cs> gcj -S Hello.java -o hello.s
cs> gcj -S Hello.java -o hello.s

_gcj compiles Java to native code_

_ZN5Hello4mainEP6JArrayIPN4java4lang6StringE:
  !_#PROLOGUE# 0
  save %sp, -128, %sp
  .LLCFI0:
    !_#PROLOGUE# 1
    st %i0, [%fp+68]
  .LLBB2:
    sethi %hi(_ZN4java4lang6System6class$E), %g1
    or %g1, %lo(_ZN4java4lang6System6class$E), %g1
    mov 1, %o4
    stb %o4, [%fp-18]
    ldub [%g1+90], %g1
    sll %g1, 24, %g1
    sra %g1, 24, %g1
    cmp %g1, 14
    bge .LL2
    nop ...
...
Same kind of assembler output, ELF file, etc, etc.
There are two translation tools in the Sun JDK.

javac  java

compiler?  JVM
Same program again.

```java
public class HelloWorld {

    public static void main (String args[]) {
        System.out.println ("Hello World!");
    }
}
```
The output of the `javac` is a binary file known as a class file. This file contains the programming instructions in what is known as byte code.
Class file format
You can convert a class back to mnemonics to get an idea of what information is in the class file.

```
> javap -c HelloWorld

class HelloWorld extends java.lang.Object {
  HelloWorld();
    0: aload_0
    1: invokespecial #1; //Method java/lang/Object."<init>":()V
    4: return

  public static void main(java.lang.String[]);
    0: getstatic    #2; //Field java/lang/System.out:Ljava/io/PrintStream;
    3: ldc         #3; //String Hello World!
    5: invokevirtual #4; //Method java/io/PrintStream.println:(Ljava/lang/String;)V
    8: return
}
```
Java virtual machine 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 (e.g., `new, putfield`)
- Operand stack management (e.g., `swap, dup2`)
- Control transfer (e.g., `ifeq, goto`)
- Method invocation and return (e.g., `invokespecial, areturn`)
Virtual machine instructions have the advantage of being portal (because it is relatively easy to write a virtual machine, and virtual impossible to translate a set of machine instructions into the machine instructions of another kind of machine.)

```plaintext
GET /~ryan/hello.html HTTP/1.0

browser (client)

HTTP/1.0 200 Document follows
Content-type: text/html
<html>
  <applet src=HelloApplet.class>
  </applet>
</html>

GET /~ryan/HelloApplet.class HTTP/1.0

WWW server

HTTP/1.0 200 Document follows
Content-type: application/octet-stream
Java byte-code
```
Although Java can be interpreted by a Java virtual machine (JVM), the byte-code could be compiled to native code. An independent, executable file may or may not be made. It is even possible to compile only some of the byte-code—the parts that are executed a lot—and not other parts.

Sun Microsystems calls the program `java` a “launcher” as details of the actions differ from typical compilers or interpreters. Such a translation/execution system is called a just-in-time (JIT) compiler, and may only compile parts of the byte-code when (and if) they are reached or executed often.

If you want Java to interpret the byte-code, you must ask for it:

```
cs> java -Xint Main
```
Do not confuse a language with its implementation.
Benchmarks mean very little.
1.0  C++ GNU g++   1.35  
1.7  Java 6 -server  2.29  
1.7  C GNU gcc      2.31  
2.3  Haskell GHC    3.14  
2.7  Intel Fortran  3.71  
2.8  Pascal Free Pascal  3.74  
3.3  C# Mono       4.44  
3.8  Ada 2005 GNAT  5.09  
12   Java 6 -Xint   16.03  
17   Smalltalk VisualWorks  23.12  
26   Python       35.43  
33   Mozart/Oz     44.62  
44   Perl         59.81  
51   PHP          68.79  
77   Ruby         104.01  

Computer Language Benchmarks Game. January 2009. Platform: Ubuntu, 2.4Ghz Intel Q6600 quad-core. First number is ratio to GNU C++ of the third column: geometric mean of the measure for the language to the best measurement for any language over all 11 benchmarks.