Many High-Level Languages

Choosing a programming language...

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

Java is often used as a teaching language because it has fewer ways "to shoot yourself in the foot."

Implementation

A program in a high-level programming language must be prepared for execution, because these langauges are designed to accomodate humans and not be executed directly by the hardware. This translation is too complex, tedious and error prone to be done manually. In fact other computer programs translate the program to instructions a machine can understand. Each high-level programming language has one or more translators or implementations which translates all programs in that language.

For example, there are the GNU `gfortran` and `g77` compilers, not to mention many commercial compilers for Fortran. Also, there are the Sun/Oracle JDK tools for Java, the IBM Jikes compiler for Java (no longer being maintained), and the GNU `gcj` compiler (also no longer current).

Terminology

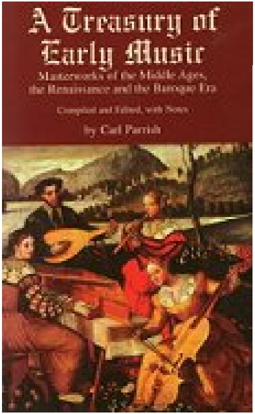
- compiling
- translating
- batch processing
- interpreting
- interactive
- just-in-time compiling

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 Compilers sometimes produce an object module which can then then be executed again and again on different data.
- batch processing: execution of a series of programs ("jobs") on a computer without manual intervention. This is in contrast to "online" or interactive programs which prompt the user for such input.
- interpreting: running the program under the control of a software program
- interactive: read-eval-print loop. Evaluate often means interpret; but could be extended to compile and run.
- just-in-time compiling: a hybrid approach of translation during execution to machine code or software emulation whichever is predicted to make execution faster

Compiler

In its usual English meaning, a compiler is one that collects and edits material written by others into a collection.



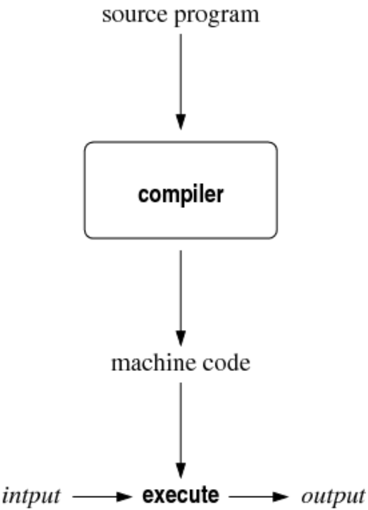
. . . compiled by Carl Parrish, . . . edited by F. Bauer and J. Eickel

Compiler

> *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.*

Friedrich L. Bauer, "Historical remarks on compiler construction", in *Compiler Construction: An Advanced Course*, edited by F. L. Bauer and Jürgen Eickel, Lecture Notes in Computer Science #21, Springer-Verlag, Berlin, pages 603-621, 1974.

Traditional Compilation



Traditional Compilation

A program — software written by people like you – translates the high-level language into a form the computer can execute.

The source program — a text file — is the input, and the output is an an executable file for some machine.

How do you write a compiler?

How do you solve a large problem?

How do you write a compiler?

How do you solve a large problem?

One important approach is to break it into well-defined sub-problems.
(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 (a dangerous facility) are found in C and C++. Java does a little preprocessing to translate character sets and Unicode escapes.

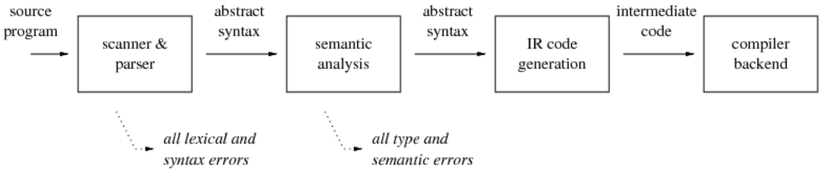
More Detailed

Input: source program

1. lexical analysis
2. syntax analysis
3. intermediate code generation
4. code optimization
5. assembly code generation

Output: Assembly program identical to the input.

Phases of the Typical Compiler



Phases of the Typical Compiler

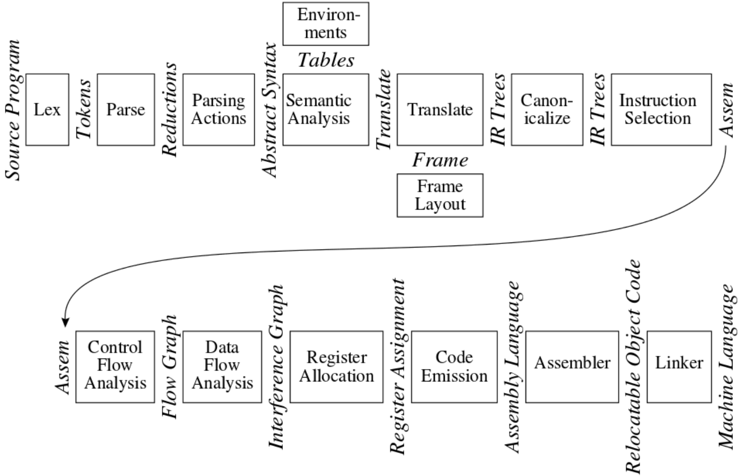


FIGURE 1.1. Phases of a compiler, and interfaces between them.

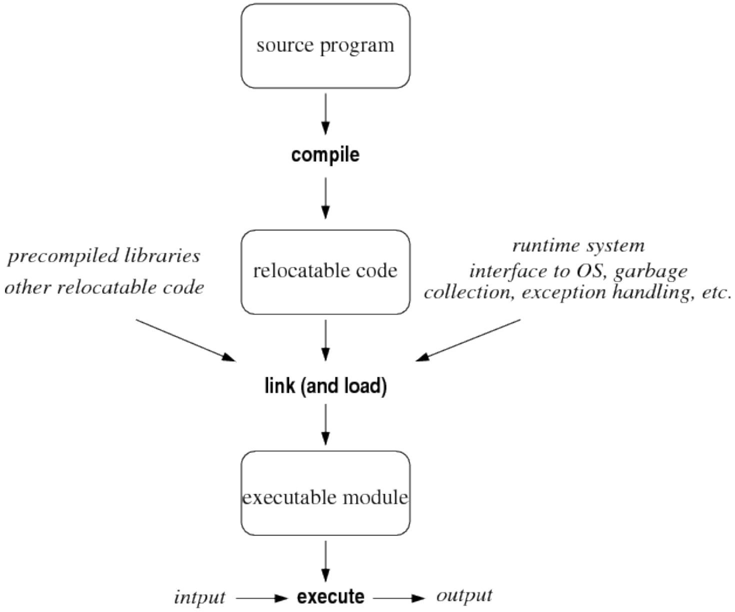
Language Systems

Language translation and execution systems are big and complex, 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).

Let us take a brief look at some of this individual steps.

Assembly


source program
compile
relocatable code
precompiled libraries
other relocatable code
runtime system
interface to OS, garbage
collection, exception handling, etc.
link (and load)
executable module
intput
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 run-time 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 implementers and the use of the underlying machine.

The key aspect of an interpreter is emulation. The key aspect of a run-time system is support of functionality.

Interpreting (continued)

Superficially, we equate *abstract* and *virtual* machine.

Technically, *abstract* connotes 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.

Run-time system

Modern, high-level languages require that a program have additional support during execution. This is sometimes called the run-time system. The run-time 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 run-time 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 run-time system may require assistance of the translation system, for example, to insert reference counting code, debugging code, etc. The run-time system must be available to every program in the language so it can run correctly, but none of the functionality might actually be used.

Run-time system

The distinction between the run-time system and the standard libraries is not always clear. Take these two statements in Java:

```
System.out.printf ("%d %s %f", 4, this, Math.sqrt(2
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).
The run-time system may depend on detailed knowledge about the program itself and the hardware. A library routine usually depends on just its arguments.

A language with a small run-time system like C, is efficient in time and space, but provides less of a virtual platform to support the programmer.

The programmer needs to write more code and know more about the hardware, but may be able to utilize the hardware more directly.

Back to translation . . .

Some Important Unix Development Tools

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

- assembler – like a compiler, a translator from source code to target code; it converts symbolic machine code to binary machine code, and from symbolic data to binary data.
- linker – combines one or more program object files and probably some library objects files into one executable file.
- loader – An integrated part of the operating system (which makes it essentially invisible) which loads the content of an executable code file into memory.

Compilation — gcc

```c
#include <stdio.h>

int
main () {
    fputs ("Hello world!\n", stdout);
    return 0;
}
```

Compilation — gcc

```
%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
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/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/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/cr
GNU ld version 2.11.2 (with BFD 2.11.2)
  Supported emulations:
   elf32_sparc
```

`crt1.o` is the main program.

`crt0` (also known as `c0`) is a set of startup routines linked into a C program that performs any initialization work required before calling the program's main function. It generally takes the form of an object file called crt0.o, often written in assembly language, which is automatically included by the linker into every executable file it builds.
`crt0` contains the most basic parts of the runtime library. As such, the exact work it performs depends on the program's compiler, operating system and C standard library implementation. Beside the initialization work required by the environment and tool chain, crt0 can perform additional operations defined by the programmer, such as executing C++ global constructors and C functions carrying the GCC's ((constructor)) attribute.
"crt" stands for "C runtime", and the zero stands for "the very beginning". However, when programs are compiled using GCC, it is also used for languages other than C. Alternative versions of crt0 are available for special usage scenarios; for example, the profiler gprof requires its programs to be compiled with `gcrt0`.

```
cs> gcc -S hello.c -o hello.s
```

```
cs> gcc -S hello.c -o hello.s

        .file   "hello.c"
gcc2_compiled.:
.section        ".rodata"
        .align 8
.LLC0:
        .asciz  "Hello world!\n"
.section        ".text"
        .align 4
        .global 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    fputs, 0
        nop
        mov     0, %i0
        b       .LL2
        nop
```

gcc compiles C to native code

ELF Executable File

```
7f 45 4c 46 01 01 01 00 00 00 00 00 00 00 00 00 01 00 03 00
01 00 00 00 00 00 00 00 00 00 00 00 fc 00 00 00 00 00 00 00
34 00 00 00 00 00 28 00 0b 00 08 00 8d 4c 24 04 83 e4 f0 ff
71 fc 55 89 e5 51 83 ec 14 a1 00 00 00 00 89 44 24 0c c7 44
24 08 0d 00 00 00 c7 44 24 04 01 00 00 00 c7 04 24 00 00 00
00 e8 fc ff ff ff b8 00 00 00 00 83 c4 14 59 5d 8d 61 fc c3
48 65 6c 6c 6f 20 77 6f 72 6c 64 21 0a 00 00 47 43 43 3a 20
28 55 62 75 6e 74 75 20 34 2e 33 2e 32 2d 31 75 62 75 6e 74
75 31 32 29 20 34 2e 33 2e 32 00 00 2e 73 79 6d 74 61 62 00
2e 73 74 72 74 61 62 00 2e 73 68 73 74 72 74 61 62 00 2e 72
65 6c 2e 74 65 78 74 00 2e 64 61 74 61 00 2e 62 73 73 00 2e
72 6f 64 61 74 61 00 2e 63 6f 6d 6d 65 6e 74 00 2e 6e 6f 74
65 2e 47 4e 55 2d 73 74 61 63 6b 00 00 00 00 00 00 00 00 00
00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
00 00 00 00 00 00 00 00 00 00 00 00 1f 00 00 00 01 00 00 00
06 00 00 00 00 00 00 00 34 00 00 00 44 00 00 00 00 00 00 00
00 00 00 00 04 00 00 00 00 00 00 00 1b 00 00 00 09 00 00 00
00 00 00 00 00 00 00 00 80 03 00 00 18 00 00 00 09 00 00 00
01 00 00 00 04 00 00 00 08 00 00 00 25 00 00 00 01 00 00 00
03 00 00 00 00 00 00 00 78 00 00 00 00 00 00 00 00 00 00 00
00 00 00 00 04 00 00 00 00 00 00 00 2b 00 00 00 08 00 00 00
03 00 00 00 00 00 00 00 78 00 00 00 00 00 00 00 00 00 00 00
00 00 00 00 04 00 00 00 00 00 00 00 30 00 00 00 01 00 00 00
02 00 00 00 00 00 00 00 78 00 00 00 0e 00 00 00 00 00 00 00
00 00 00 00 01 00 00 00 00 00 00 00 38 00 00 00 01 00 00 00
00 00 00 00 00 00 00 00 86 00 00 00 25 00 00 00 00 00 00 00
00 00 00 00 01 00 00 00 00 00 00 00 41 00 00 00 01 00 00 00
00 00 00 00 00 00 00 00 ab 00 00 00 00 00 00 00 00 00 00 00
00 00 00 00 01 00 00 00 00 00 00 00 11 00 00 00 03 00 00 00
00 00 00 00 00 00 00 00 ab 00 00 00 51 00 00 00 00 00 00 00
00 00 00 00 01 00 00 00 00 00 00 00 01 00 00 00 02 00 00 00
00 00 00 00 00 00 00 00 b4 02 00 00 b0 00 00 00 0a 00 00 00
08 00 00 00 04 00 00 00 10 00 00 00 09 00 00 00 03 00 00 00
00 00 00 00 00 00 00 00 64 03 00 00 1c 00 00 00 00 00 00 00
00 00 00 00 01 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
```

Executable and Linkable Format (ELF, formerly called Extensible Linking Format) is a common standard file format for executables, object code, shared libraries, and core dumps. First published in the System V Application Binary Interface specification, and later in the Tool Interface Standard, it was quickly accepted among different vendors of Unix systems. In 1999 it was chosen as the standard binary file format for Unix and Unix-like systems on x86 by the 86open project. It has replaced a.out and COFF formats in Unix-like operating systems.

ELF is flexible and extensible, and it is not bound to any particular processor or architecture. This has allowed it to be adopted by many different operating systems on many different platforms.

Compilation

```
0000 7f45 4c46 0102 0100 0000 0000 0000 0000    del   E   L   F soh stx soh nul nul nul nul nul nul nul nul n
0020 0001 0002 0000 0001 0000 0000 0000 0000    nul soh nul stx nul nul nul soh nul nul nul nul nul nul nul n
0040 0000 00e8 0000 0000 0034 0000 0000 0028    nul nul nul   h nul nul nul nul nul   4 nul nul nul nul nul
0060 000a 0007 9de3 bf90 1300 0000 9012 6000    nul  nl nul bel  gs   c   ? dle dc3 nul nul nul dle dc2   ` n
0100 1500 0000 9212 a000 4000 0000 0100 0000    nak nul nul nul dc2 dc2  sp nul   @ nul nul nul soh nul nul n
0120 b010 2000 1080 0002 0100 0000 81c7 e008      0 dle  sp nul dle nul nul stx soh nul nul nul soh   G   `
0140 81e8 0000 0000 0000 4865 6c6c 6f20 776f    soh   h nul nul nul nul nul nul   H   e   l   l   o  sp   w
0160 726c 6421 0a00 0000 0047 4343 3a20 2847      r   l   d   !  nl nul nul nul nul   G   C   C   :  sp   (
0200 4e55 2920 322e 3935 2e33 2032 3030 3130      N   U   )  sp   2   .   9   5   .   3  sp   2   0   0   1
0220 3331 3520 2872 656c 6561 7365 2900 002e      3   1   5  sp   (   r   e   l   e   a   s   e   ) nul nul
0240 7379 6d74 6162 002e 7374 7274 6162 002e      s   y   m   t   a   b nul   .   s   t   r   t   a   b nul
0260 7368 7374 7274 6162 002e 7465 7874 002e      s   h   s   t   r   t   a   b nul   .   t   e   x   t nul
0300 7265 6c61 2e74 6578 7400 2e64 6174 6100      r   e   l   a   .   t   e   x   t nul   .   d   a   t   a n
0320 2e62 7373 002e 726f 6461 7461 002e 636f      .   b   s   s nul   .   r   o   d   a   t   a nul   .   c
0340 6d6d 656e 7400 0000 0000 0000 0000 0000      m   m   e   n   t nul nul nul nul nul nul nul nul nul nul n
0360 0000 0000 0000 0000 0000 0000 0000 0000    nul nul nul nul nul nul nul nul nul nul nul nul nul nul nul n
*
0420 0000 001b 0000 0001 0000 0006 0000 0000    nul nul nul esc nul nul nul soh nul nul nul ack nul nul nul n
0440 0000 0034 0000 0030 0000 0000 0000 0000    nul nul nul   4 nul nul nul   0 nul nul nul nul nul nul nul n
0460 0000 0004 0000 0000 0000 0021 0000 0004    nul nul nul eot nul nul nul nul nul nul nul   ! nul nul nul e
0500 0000 0000 0000 0000 0000 0368 0000 003c    nul nul nul nul nul nul nul nul nul nul etx   h nul nul nul
0520 0000 0008 0000 0001 0000 0004 0000 000c    nul nul nul  bs nul nul nul soh nul nul nul eot nul nul nul
0540 0000 002c 0000 0001 0000 0003 0000 0000    nul nul nul   , nul nul nul soh nul nul nul etx nul nul nul n
0560 0000 0064 0000 0000 0000 0000 0000 0000    nul nul nul   d nul nul nul nul nul nul nul nul nul nul nul n
0600 0000 0001 0000 0000 0000 0032 0000 0008    nul nul nul soh nul nul nul nul nul nul nul   2 nul nul nul
0620 0000 0003 0000 0000 0000 0064 0000 0000    nul nul nul etx nul nul nul nul nul nul nul   d nul nul nul n
0640 0000 0000 0000 0000 0000 0001 0000 0000    nul nul nul nul nul nul nul nul nul nul nul soh nul nul nul n
0660 0000 0037 0000 0001 0000 0002 0000 0000    nul nul nul   7 nul nul nul soh nul nul nul stx nul nul nul n
0700 0000 0068 0000 0010 0000 0000 0000 0000    nul nul nul   h nul nul nul dle nul nul nul nul nul nul nul n
```

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:
 0000 48656c6c 6f20776f 726c6421 0a000000  Hello world!....
Contents of section .comment:
 0000 00474343 3a202847 4e552920 322e3935  .GCC: (GNU) 2.95
 0010 2e332032 30303130 33313520 2872656c  .3 20010315 (rel
 0020 65617365 2900                        ease).
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
```

Compilation

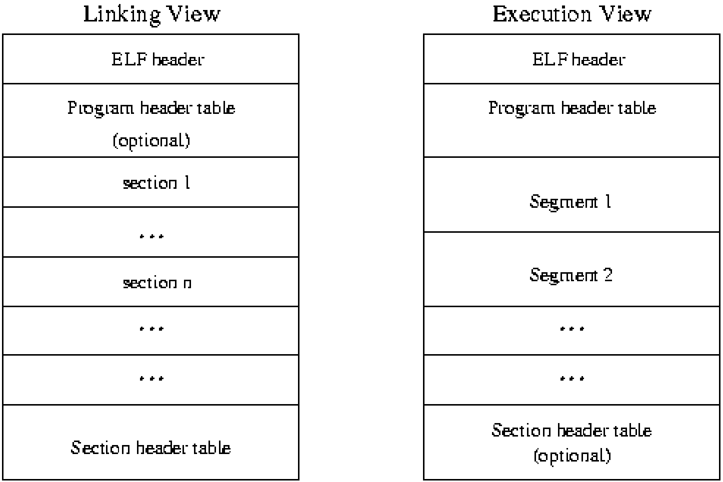
```
hello.o:     file format elf32-i386

Contents of section .text:
 0000 8d4c2404 83e4f0ff 71fc5589 e55183ec  .L$.....q.U..Q..
 0010 14a10000 00008944 240cc744 24080d00  .......D$..D$...
 0020 0000c744 24040100 0000c704 24000000  ...D$.......$...
 0030 00e8fcff ffffb800 00000083 c414595d  ..............Y]
 0040 8d61fcc3                             .a..
Contents of section .rodata:
 0000 48656c6c 6f20776f 726c6421 0a00      Hello world!..
Contents of section .comment:
 0000 00474343 3a202855 62756e74 7520342e  .GCC: (Ubuntu 4.
 0010 332e322d 31756275 6e747531 32292034  3.2-1ubuntu12) 4
 0020 2e332e32 00                          .3.2.

Disassembly of section .text:

00000000 <main>:
   0: 8d 4c 24 04             lea    0x4(%esp),%ecx
   4: 83 e4 f0                and    $0xfffffff0,%esp
   7: ff 71 fc                pushl  -0x4(%ecx)
   a: 55                      push   %ebp
   b: 89 e5                   mov    %esp,%ebp
   d: 51                      push   %ecx
   e: 83 ec 14                sub    $0x14,%esp
  11: a1 00 00 00 00          mov    0x0,%eax
  16: 89 44 24 0c             mov    %eax,0xc(%esp)
  1a: c7 44 24 08 0d 00 00    movl   $0xd,0x8(%esp)
  21: 00
  22: c7 44 24 04 01 00 00    movl   $0x1,0x4(%esp)
  29: 00
  2a: c7 04 24 00 00 00 00    movl   $0x0,(%esp)
  31: e8 fc ff ff ff          call   32 <main+0x32>
  36: b8 00 00 00 00          mov    $0x0,%eax
  3b: 83 c4 14                add    $0x14,%esp
```

ELF – Executable and Linkable Format

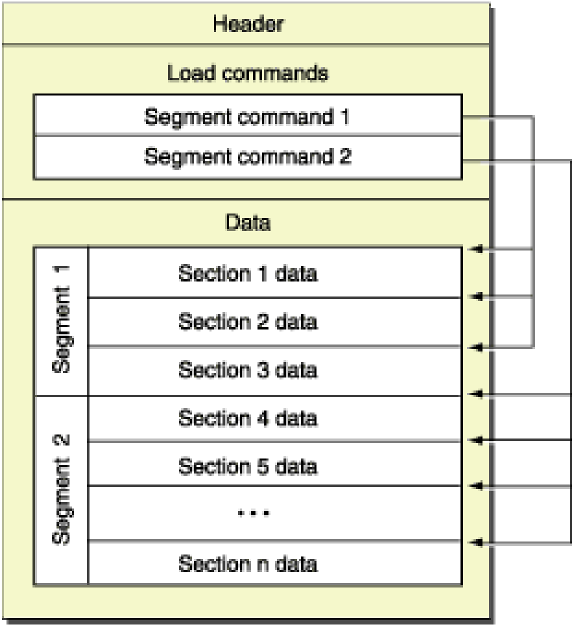


ELF – Executable and Linkable Format

```
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, Motorol, MIPS, ...   */
   uint32_t      e_version;
   uintN_t       e_entry;     /* entry point */
   ...
} ElfN_Ehdr;
```

Note `objdump` (GNU/Linux), `readelf` (Unix), and `elfdump` (Solaris) all view elf files. Note `otool` (Darwin) to view Mach-o files.

Mach-O


Header
Load commands
Segment command 1
Segment command 2
Data
Segment 1
Section 1 data
Section 2 data
Section 3 data
Segment 2
Section 4 data
Section 5 data
...
Section n data

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;       /* ovject, 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) */
```

The traditional compiler produces machine instructions to be executed by the CPU.

The traditional compiler produces an executable file which can be used over and over again.

The traditional compiler links in all the support code. (With dynamic linking the additional code might not be a part of the initial executable file, but might be added while the program is running.)

Translating Java

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

1. IBM Jikes
2. GNU gcj
3. Sun/Oracle Java 2 SDK

Translating Java

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

1. IBM Jikes
2. GNU gcj
3. Sun/Oracle Java 2 SDK

We begin by looking at GNU gcj to see a traditional translator in action. Then we move to the Sun/Oracle Java 2 SDK and see the important role of byte code.

Compilation — gcj

```
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=/
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-r
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/cc1 /var/tmp//ccWJ2hCQ.i -quiet -dumpbase Hellom
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 he
GNU ld version 2.14 20030612
  Supported emulations:
   elf32_sparc
   elf64_sparc
```

`jc1` does the translation (and preprocessing) of the Java source code into assembly code.

The main program is generated in the C programming language by `jvgemain`.

```
cs> gcj -S Hello.java -o hello.s
```

```
cs> gcj -S Hello.java -o hello.s

_ZN5Hello4mainEP6JArrayIPN4java4lang6StringEE:
        !#PROLOGUE# 0
        save    %sp, -128, %sp
.LLCFI0:
        !#PROLOGUE# 1
        st      %i0, [%fp+68]
.LLBB2:
        sethi   %hi(_ZN4java4lang6System6class$E), %g1
        or      %g1, %lo(_ZN4java4lang6System6class$E),
        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
        ...
```

gcj compiles Java to native code

Same kind of assembler output, ELF file, etc, etc.

The point is that the traditional compiler produces machine instructions to be executed by the CPU.

The traditional compiler produces an executable file (object module) which can be used over and over again.

The traditional compiler links in all the support code. (With dynamic linking the additional code might not be a part of the initial executable file, but might be added while the program is running.)

Translators, assemblers, and linkers are just program. They are not magic. You can write one, if you understand what the input is and what the output is.

Translation — Sun/Oracle JDK

There are two translation tools in the Sun/Oracle JDK.

`javac` `java`

compiler? JVM

Translation — Sun JDK

Same program again.

```
public class HelloWorld  {

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

}
```

Translation — Sun JDK

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.

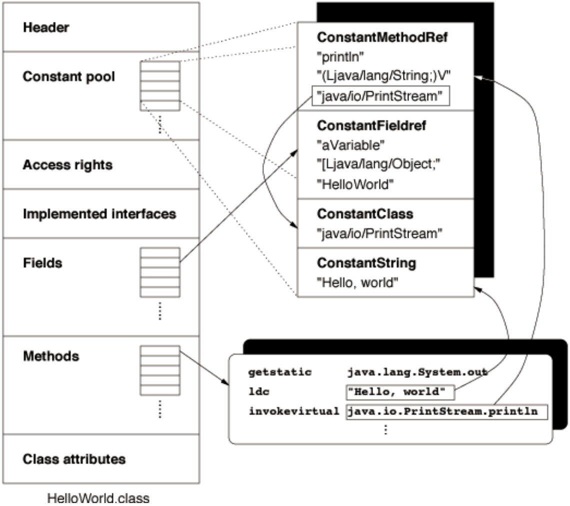
```
000  ca fe ba be 00 00 00 31 00 1a 0a 00 06 00 0c 09 00 0d
018  00 0e 08 00 0f 0a 00 10 00 11 07 00 12 07 00 13 01 00
036  06 3c 69 6e 69 74 3e 01 00 03 28 29 56 01 00 04 43 6f
054  64 65 01 00 04 6d 61 69 6e 01 00 16 28 5b 4c 6a 61 76
072  61 2f 6c 61 6e 67 2f 53 74 72 69 6e 67 3b 29 56 0c 00
090  07 00 08 07 00 14 0c 00 15 00 16 01 00 0c 48 65 6c 6c
108  6f 20 57 6f 72 6c 64 21 07 00 17 0c 00 18 00 19 01 00
126  0a 48 65 6c 6c 6f 57 6f 72 6c 64 01 00 10 6a 61 76 61
144  2f 6c 61 6e 67 2f 4f 62 6a 65 63 74 01 00 10 6a 61 76
162  61 2f 6c 61 6e 67 2f 53 79 73 74 65 6d 01 00 03 6f 75
180  74 01 00 15 4c 6a 61 76 61 2f 69 6f 2f 50 72 69 6e 74
198  53 74 72 65 61 6d 3b 01 00 13 6a 61 76 61 2f 69 6f 2f
216  50 72 69 6e 74 53 74 72 65 61 6d 01 00 07 70 72 69 6e
234  74 6c 6e 01 00 15 28 4c 6a 61 76 61 2f 6c 61 6e 67 2f
252  53 74 72 69 6e 67 3b 29 56 00 20 00 05 00 06 00 00 00
270  00 00 02 00 00 00 07 00 08 00 01 00 09 00 00 00 11 00
288  01 00 01 00 00 00 05 2a b7 00 01 b1 00 00 00 00 00 09
306  00 0a 00 0b 00 01 00 09 00 00 00 15 00 02 00 01 00 00
324  00 09 b2 00 02 12 03 b6 00 04 b1 00 00 00 00 00 00
```

Class file format

Note the magic number of a class file “cafebabe.”

There are two meanings for the phrase magic number.

Class file format



Translation — Sun/Oracle JDK

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
}
```

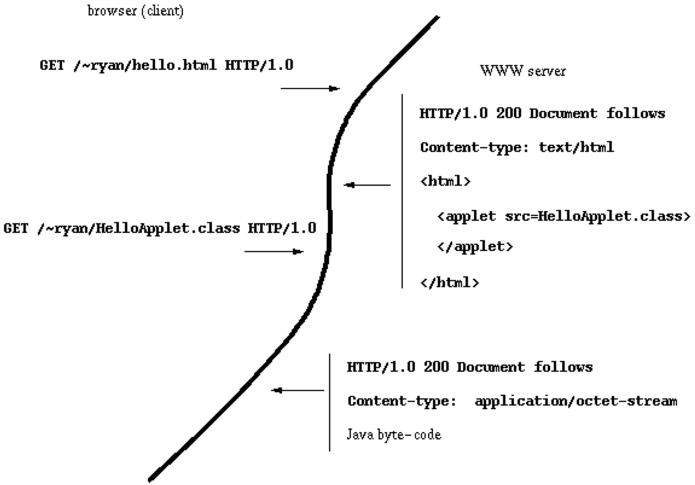
Java virtual machine instructions:

- 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.)

A class file is machine independent, like a PNG or JPG file.

Java gained wide-spread notice in the 1990s by providing the first mechanism for dynamic content on the WWW: applets.



Java – Just-In-Time

Although the byte-code of a Java program can be interpreted, the byte-code could be just as well 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 does not have a good name — a virtual machine, perhaps — This hybrid interpreter/compiler may only compile parts of the byte-code by a JIT compiler 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
```

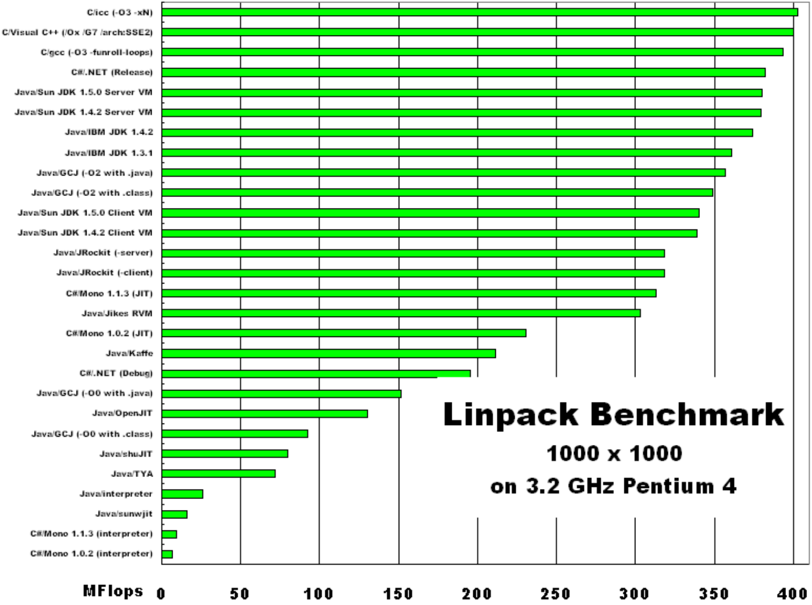
Unlike the traditional compiler, the JIT compiler does not begin compiling to native code until the user of the program launches execution!

Compilation of programs is so fast these days that the user does not usually mind the extra execution time devoted to compilation. (If the program is run by the developer and modified frequently, the total amount of time might even be less than the traditional compilation approach.)

Furthermore, `java` does not even look for the class files containing the byte code to translate until after the user launches the programs. This make Java difficult to deploy as the user my be uncertain if all the class files are available when the program is launched.

Do not confuse a language with its implementation.

Benchmarks mean very little.


C/icc (-O3 -xN)
C/Visual C++ (/Ox /G7 /arch:SSE2)
C/gcc (-O3 -funroll-loops)
C#/.NET (Release)
Java/Sun JDK 1.5.0 Server VM
Java/Sun JDK 1.4.2 Server VM
Java/IBM JDK 1.4.2
Java/IBM JDK 1.3.1
Java/GCJ (-O2 with .java)
Java/GCJ (-O2 with .class)
Java/Sun JDK 1.5.0 Client VM
Java/Sun JDK 1.4.2 Client VM
Java/JRockit (-server)
Java/JRockit (-client)
C#/Mono 1.1.3 (JIT)
Java/Jikes RVM
C#/Mono 1.0.2 (JIT)
Java/Kaffe
C#/.NET (Debug)
Java/GCJ (-O0 with .java)
Java/OpenJIT
Java/GCJ (-O0 with .class)
Java/shuJIT
Java/TYA
Java/interpreter
Java/sunwjit
C#/Mono 1.1.3 (interpreter)
C#/Mono 1.0.2 (interpreter)
Linpack Benchmark
1000 x 1000
on 3.2 GHz Pentium 4
MFlops
0
50
100
150
200
250
300
350
400

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.