We wish to focus now on the translation of a high-level language.
Choosing a programming language...

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

BTW in the CSE curriculum we choose to teach programming using Java for several reasons, but primarily because it has fewer ways “to shoot yourself in the foot.”
A program in a high-level programming language must be prepared for execution, because there is no computer that understands Java instructions directly. This preparation 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 its own translator or implementation which translates all programs in that language. These programs are often compilers.

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/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JDK tools for Java, the IBM Jikes compiler for Java (no longer being maintained), and the GNU gcj compiler (also no longer current).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
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
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.

I like to use the vague term *translator*.
Traditional Compilation

source program

compiler

machine code

input → execute → output
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 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 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 and Unicode escapes.
More Detailed
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).

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

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 run-time support these languages expect.
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.
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.
The distinction between the run-time 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).

The run-time system may depend on detailed knowledge about the very program itself. A library routine usually depends on just its arguments.

A language a small run-time system like C, is efficient in time and space, but provides less of a virtual platform to support the programmer.
Back to translation ...
Important Unix 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.
```
#include <stdio.h>

int main () {
    fputs ("Hello world!\n", stdout);
    return 0;
}
```
source code — assembly instructions — relocatable module — executable module
C — SPARC instructions — ELF — ELF
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.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-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-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
crt1.o is the main program.
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**
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

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 2872656l .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 0e8fcff ffff800 00000083 c414595d ............Y]
0040 8d61fc33 .a..
```

Contents of section .rodata:

```
0000 48656c6c 6f20776f 726c6421 0a00 Hello world!..
```

Contents of section .comment:

```
0000 00474343 202855 62756e74 7520342e .GCC: (Ubuntu 4.
0010 332e322d 31756275 6e747531 32292034 3.2-1ubuntu12) 4
0020 3.2.
```

Disassembly of section .text:

```
00000000 <main>:
  0: 8d 4c 24 04 lea 0x4(%esp),%ecx
  4: 83 e4 f0 and $0xfffffffff0,%esp
  7: ff 71 fc pushl -0x4(%ecx)
  a: 55 push %ebp
  b: 89 e5 mov %esp,%ebp
  d: 83 ec 14 sub $0x14,%esp
  11: a1 00 00 00 00 mov 0x0,%eax
  16: 89 44 24 0c mov %eax,%ecx
  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

**Linking View**

<table>
<thead>
<tr>
<th>ELF header</th>
</tr>
</thead>
<tbody>
<tr>
<td>Program header table</td>
</tr>
<tr>
<td>(optional)</td>
</tr>
<tr>
<td><strong>section 1</strong></td>
</tr>
<tr>
<td><strong>...</strong></td>
</tr>
<tr>
<td><strong>section n</strong></td>
</tr>
<tr>
<td><strong>...</strong></td>
</tr>
<tr>
<td>Section header table</td>
</tr>
</tbody>
</table>

**Execution View**

<table>
<thead>
<tr>
<th>ELF header</th>
</tr>
</thead>
<tbody>
<tr>
<td>Program header table</td>
</tr>
<tr>
<td><strong>Segment 1</strong></td>
</tr>
<tr>
<td><strong>Segment 2</strong></td>
</tr>
<tr>
<td><strong>...</strong></td>
</tr>
<tr>
<td><strong>...</strong></td>
</tr>
<tr>
<td>Section header table</td>
</tr>
<tr>
<td>(optional)</td>
</tr>
</tbody>
</table>
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 all Note objdump (GNU/Linux), readelf (Unix), and elfdump (Solaris) view elf files. Note otool (Darwin) to view Mach-o files.
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) */
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.
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/cc1 /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 hello ...
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

gcj compiles Java to native code
cs> gcj -S Hello.java -o hello.s

_gcj compiles Java to native code_.

```
_ZN5Hello4mainEP6JArrayIPN4java4lang6StringEF:
  !#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
  ...
```
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.)
There are two translation tools in the Sun/Oracle 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.
Note the magic number of a class file “cafebabe.”
There are two meanings for the phrase magic number.
Class file format

Header
Constant pool
Access rights
Implemented interfaces
Fields
Methods
Class attributes

ConstantMethodRef
"println"
"(Ljava/lang/String;)V"
"java/io/PrintStream"

ConstantFieldref
"aVariable"
"[Ljava/lang/Object;"
"HelloWorld"

ConstantClass
"java/io/PrintStream"

ConstantString
"Hello, world"

getstatic java.lang.System.out
ldc "Hello, world"
invokevirtual java.io.PrintStream.println
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.
Although Java 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 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:

```bash
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.
<table>
<thead>
<tr>
<th>Ratio</th>
<th>Language</th>
<th>Ratio</th>
</tr>
</thead>
<tbody>
<tr>
<td>1.0</td>
<td>C++ GNU g++</td>
<td>1.35</td>
</tr>
<tr>
<td>1.7</td>
<td>Java 6 -server</td>
<td>2.29</td>
</tr>
<tr>
<td>1.7</td>
<td>C GNU gcc</td>
<td>2.31</td>
</tr>
<tr>
<td>2.3</td>
<td>Haskell GHC</td>
<td>3.14</td>
</tr>
<tr>
<td>2.7</td>
<td>Intel Fortran</td>
<td>3.71</td>
</tr>
<tr>
<td>2.8</td>
<td>Pascal Free Pascal</td>
<td>3.74</td>
</tr>
<tr>
<td>3.3</td>
<td>C# Mono</td>
<td>4.44</td>
</tr>
<tr>
<td>3.8</td>
<td>Ada 2005 GNAT</td>
<td>5.09</td>
</tr>
<tr>
<td>12</td>
<td>Java 6 -Xint</td>
<td>16.03</td>
</tr>
<tr>
<td>17</td>
<td>Smalltalk VisualWorks</td>
<td>23.12</td>
</tr>
<tr>
<td>26</td>
<td>Python</td>
<td>35.43</td>
</tr>
<tr>
<td>33</td>
<td>Mozart/Oz</td>
<td>44.62</td>
</tr>
<tr>
<td>44</td>
<td>Perl</td>
<td>59.81</td>
</tr>
<tr>
<td>51</td>
<td>PHP</td>
<td>68.79</td>
</tr>
<tr>
<td>77</td>
<td>Ruby</td>
<td>104.01</td>
</tr>
</tbody>
</table>

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.