Levels of Programming Languages

Gerald Penn
CSC 324
Levels of Programming Language

• Microcode
• Machine code
• Assembly Language
• Low-level Programming Language
• High-level Programming Language
Levels of Programming Language

• Microcode
  – Machine-specific code that directs the individual components of a CPU’s data-path to perform small-scale operations.
  – CPU: central processing unit of a computer, typically consisting of:
    • Control unit
    • Arithmetic/logical unit (ALU)
    • Registers – high-speed memory locations to store temporary results and control information. Foremost among these is the program counter, which points to the next instruction to be executed.
  – The CPU is connected to I/O devices and main memory by parallel channels called buses.
Levels of Programming Language

• Microcode
  – Machine-specific code that directs the individual components of a CPU’s data-path to perform small-scale operations.
  – Data-path: the ALU, its inputs and outputs.
  – People who build computers program in micro-code. The programs that you write are converted (as explained later) into machine code.
  – Every machine code instruction tells the CPU to execute a certain microprogram, written in micro-code.
  – Often these programs are implemented in hardware.
  – On the other hand, some microprocessors are:
    • programmable, e.g., many digital signal processing chips that mobile telephones use, FPGAs, or
    • reconfigurable – they can actually rewire themselves.
Levels of Programming Language

• Machine code / Assembly Language
  – Machine code instructions still depend on the computer’s architecture, but the variation isn’t as great; many CPUs manufactured around the same time or by the same company will use the same machine code sets, in fact.
  – Assembly language is a symbolic presentation of machine code so that people (very dedicated people with lots of free time) can read programs written in it.
  – Most assemblers (programs that convert assembly code to machine code) support labelling and macros to make assembly language programming easier.
  – Some recent assemblers support looping control structures, simple data structures and even types!
<table>
<thead>
<tr>
<th>Address</th>
<th>Label</th>
<th>Instruction</th>
<th>Object Code</th>
</tr>
</thead>
<tbody>
<tr>
<td>2048</td>
<td></td>
<td>.begin</td>
<td></td>
</tr>
<tr>
<td>2048</td>
<td></td>
<td>.org 2048</td>
<td></td>
</tr>
<tr>
<td>2048</td>
<td>a_start</td>
<td>.equ 3000</td>
<td></td>
</tr>
<tr>
<td>2064</td>
<td></td>
<td>ld length,%</td>
<td></td>
</tr>
<tr>
<td>2068</td>
<td></td>
<td>be done</td>
<td>00000010 100...</td>
</tr>
<tr>
<td>2068</td>
<td></td>
<td>addcc %r1, -4,%r1</td>
<td>10000010 100...</td>
</tr>
<tr>
<td>2072</td>
<td></td>
<td>addcc %41,%r2,%r4</td>
<td>10001000 100...</td>
</tr>
<tr>
<td>2076</td>
<td></td>
<td>ld %r4,%r5</td>
<td>11001010 000...</td>
</tr>
<tr>
<td>2080</td>
<td></td>
<td>ba loop</td>
<td>00010000 101...</td>
</tr>
<tr>
<td>2084</td>
<td></td>
<td>addcc %r3,%r5,%r3</td>
<td>10000110 100...</td>
</tr>
<tr>
<td>2088</td>
<td>done:</td>
<td>jmpl %r15+4,%r0</td>
<td>10000001 110...</td>
</tr>
<tr>
<td>2092</td>
<td>length:</td>
<td>20</td>
<td>00000000 000...</td>
</tr>
<tr>
<td>2096</td>
<td>address:</td>
<td>a_start</td>
<td></td>
</tr>
<tr>
<td>3000</td>
<td>a:</td>
<td>.org a_start</td>
<td></td>
</tr>
</tbody>
</table>
Levels of Programming Language

• Low-level Programming Language
  – Formerly known as high-level programming languages. 😊
  – e.g.: FORTRAN, COBOL, BASIC, arguably C
  – These languages have looping constructs, procedures, functions, some typing – the trappings of modern programming languages.
  – Big improvement over assembly language.
Levels of Programming Language

• High-level Programming Language
  – e.g.: Java, Python, ML, Prolog, MATLAB, etc.
  – These are very convenient, but also very far removed from the computer they are running on.
    • Type checking
    • Easier to debug
    • You may never even see a memory address.
  – As a result, they typically aren’t as efficient.
  – They still may not be portable: *implementation dependence*. Java has had some problems with this.
Compilation

• A compiler is a program that converts a program written at one of the higher levels into an equivalent program at some lower level.
  – Some people have even tried to use C as a target language for Java, ML or Prolog compilers.
  – Not always the next level down, though.
  – *Native code compilers* compile the code all the way down into the machine code level.
Compilation

• Advantages:
  – Compile once, run target many times
  – Compiler can optimize the speed of the target, even if the optimization itself takes a long time.
    • Actually, most compilers define their own intermediate code levels, and perform optimizations at the source level, the intermediate level, and at the target level. Which level is best depends on the optimization.

• Disadvantage: debugging requires much more software support
  – typically through annotated object code and IDE extensions.
Interpreted Code

• Code that isn’t compiled before execution is *interpreted*.
• Some programming languages have both compilers and interpreters.
• Not a black-and-white distinction either – it’s very rare for an interpreter to perform no compilation whatsoever
  – a *byte compiler* translates source code into a more compact form by coding keywords and hashing variables names and other strings.
Interpreted Code

• Advantages:
  – Creates the impression that your computer actually runs on a high-level language
  – Easier to provide feedback for debugging because execution proceeds from (something close to) source code
  – Easier to rapidly prototype
  – Often easier to add code while running code.

• Disadvantages:
  – Slower
  – Independent executions repeat much of the same work.
Inside your Interpreter

• The fetch-execute cycle
  – initialize the program counter
  – loop
    • fetch instruction pointed to by PC
    • increment the PC
    • decode the instruction
    • fetch data from memory, as necessary
    • execute the instruction
    • store the result
  – end loop
Inside your Interpreter

• The idea that an imperative program is sitting around executing your precious ML code is anathema to functional programmers.

• But we can think of it functionally: then it’s called the *read-eval-print loop*, a recursive program that repeatedly:
  – initializes the evaluation environment
  – reads an expression
  – evaluates the expression, and then
  – prints the expression.