

# Software Lesson 2 Outline

1. Software Lesson 2 Outline
2. Languages
3. Ingredients of a Language
4. Kinds of Languages
5. Natural Languages: How Many?
6. Natural Languages Don't Have to Arise Naturally
7. Natural Languages Can Be Flexible
8. Natural Languages Can Be Mixed Together
9. Natural Languages Can Be Ambiguous
10. Natural Languages Can Be Flexible About Correctness
11. Programming Languages
12. Natural Languages vs Programming Languages
13. Programming Language Hierarchy
14. High Level Languages
15. Assembly Languages
16. Machine Languages
17. Converting Between Languages
18. Compiler
19. Interpreter
20. Assembler
21. Our Old Friend `hello_world.c`
22. Compiler Details
23. Compiler Details (cont'd)
24. Elements of a Compiler #1
25. Elements of a Compiler #2
26. Phases of Compiling
27. Compiling a C Statement
28. Assembly Code for `hello_world.c` #1
29. Assembly Code for `hello_world.c` #2
30. Machine Code for `hello_world.c`
31. How to Program in Machine Language Directly
32. Why Not Do Everything in Machine Language?
33. Why Not Do Everything in Assembly Language?
34. The Programming Process
35. What is an Algorithm?
36. Algorithms
37. Algorithm Example: Eating a Bowl of Corn Flakes
38. Top-Down Design
39. Eating Cornflakes: Top Level



# Languages

---

- What is a language?
- Kinds of languages
  - Natural languages
  - Programming languages (also known as Formal languages)
- Converting between programming languages
  - Compilers
  - Interpreters
  - Assemblers



# Ingredients of a Language

---

- **Symbols**: a set of **words** and **punctuation** (in computing, words and punctuation are collectively known as **tokens**)
- **Grammar** (also known as **syntax**): a set of rules for putting symbols together to get valid statements
- **Semantics**: a set of rules for interpreting the **meaning** of a grammatically valid statement



# Kinds of Languages

---

- *Natural languages*: used in human communication
- *Programming languages* (also known as *formal languages*): used by computers (among others)



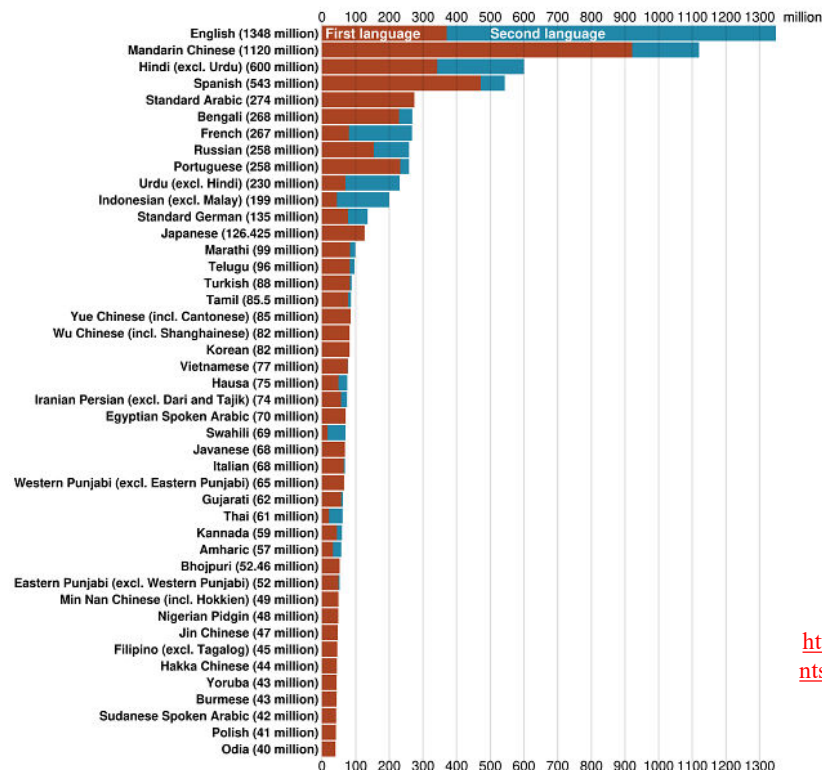
# Natural Languages: How Many?

- There are said to be 7000+ natural languages in the world:

<https://www.ethnologue.com/guides/how-many-languages>

- Examples: English, Chinese, Swahili, Navajo, Quechua, Maori

Languages by total number of speakers



[https://www.reddit.com/r/dataisbeautiful/comments/t5oitk/most\\_spoken\\_languages\\_in\\_the\\_world/?rdt=36221](https://www.reddit.com/r/dataisbeautiful/comments/t5oitk/most_spoken_languages_in_the_world/?rdt=36221)



# Natural Languages Don't Have to Arise Naturally

Not all natural languages arise naturally – some are created by people, on purpose.



[https://en.wikipedia.org/wiki/Jabba\\_the\\_Hutt](https://en.wikipedia.org/wiki/Jabba_the_Hutt)



<https://en.wikipedia.org/wiki/Worf>



<https://en.wikipedia.org/wiki/Legolas>



[https://en.wikipedia.org/wiki/Incubus\\_\(1966\\_film\)](https://en.wikipedia.org/wiki/Incubus_(1966_film))



# Natural Languages Can Be Flexible

---

Natural languages typically can be described by formal rules (grammar), but often aren't rigidly governed by these rules in everyday use:

**“Any noun can be verbed.”**

**“I might could get me one o' them there computers.”**





# Natural Languages Can Be Mixed Together

Natural languages **CAN mix words from different languages** – and even **syntax** (elements of grammar) from different languages – in a single sentence:

**“Hey, amigo, is it all right by you if I kibbitz your pachisi game while we watch your anime?”**



<https://en.wikipedia.org/wiki/Parcheesi>



<https://www.indiewire.com/gallery/the-20-most-iconic-characters-in-studio-ghibli-history/big-totoro/>





# Natural Languages Can Be Ambiguous

Natural languages CAN be ambiguous:

- **“When did he say she was going?”**

could be interpreted as:

- State the time at which he said the words “She was going.”

OR

- According to him, at what time was she going?

- **“You can’t put too much water in a nuclear reactor.”**

could be interpreted as:

- You shouldn’t put a lot of water in a nuclear reactor.

OR

- There’s no upper limit to how much water you can put in a nuclear reactor.

[https://www.onesnladay.com/wp-content/uploads/2019/02/11-17-1984\\_0.53.55.00-300x225.jpg](https://www.onesnladay.com/wp-content/uploads/2019/02/11-17-1984_0.53.55.00-300x225.jpg)



# **Natural Languages Can Be Flexible About Correctness**

---

Natural languages can have **plenty of flexibility** regarding “correctness;” for example, “ain’t,” split infinitives, ending a sentence with a preposition.

**“That is something up with which I will not put.”**



# Programming Languages

---

- Examples: C, Java, HTML, Haskell, SAS
- Also known as *formal languages*
- **Completely described** and **rigidly governed** by formal rules
- **Cannot mix** the words of multiple languages,  
or the syntax of multiple languages, in the same program
- **Cannot be ambiguous**
- Words and syntax must be **EXACTLY** correct in every way



# Natural Languages vs Programming Languages

<i><b>PROPERTY</b></i>	<i><b>NAT'L</b></i>	<i><b>PROG</b></i>
<u><b>Completely described</b></u> and <u><b>rigidly governed</b></u> by formal rules	no	<b>YES</b>
<u><b>CAN</b></u> mix the words of multiple languages, or the syntax of multiple languages, in the same program	<b>YES</b>	no
<u><b>CAN</b></u> be ambiguous	<b>YES</b>	no
Words and syntax must be <u><b>EXACTLY</b></u> correct in every way	no	<b>YES</b>



# Programming Language Hierarchy

---

- High Level Languages
- Assembly Languages
- Machine Languages



# High Level Languages

---

- **Human-readable**
- Most are **standardized**, so they can be used on just about any kind of computer.
- Examples: C, Fortran 90, Java, HTML, Haskell, SAS
- Typically, they are designed for a particular kind of application; for example:
  - C for operating system design
  - Fortran 90 for scientific & engineering applications
  - Java for embedded systems (originally designed for interactive TV)
  - HTML for hypertext (webpages)
  - SAS for statistics

But often, their uses in real life are broader their original purpose.



# Assembly Languages

---

- **Human-readable**
- **Specific to a particular CPU family**; for example:
  - Intel/AMD x86 (PCs, servers, some handhelds)
  - ARM (handhelds such as smartphones and tablets)
  - IBM POWER (server computers)
- So, for example, a program in x86 assembly language cannot be directly run on a machine with an ARM CPU.
- Set of **simple commands**; for example:
  - Load a value from a location in main memory
  - Add two numbers
  - Branch to an instruction out of sequence





# Machine Languages

---

- **Not human-readable**, except with immense effort
- Binary code that the CPU family understands directly
- Binary representation of the CPU family's assembly language



# Converting Between Languages

---

*Compilers*, *interpreters* and *assemblers* are programs that convert human-readable source code into machine-readable executable code.



# Compiler

---

- Converts a human-readable high level language source code of a program into a machine language *executable* program
- Converts an entire source code all at once
- Must be done before executing the program
- Example compiled languages: Fortran, C, C++, Pascal



# Interpreter

---

- Converts a human-readable high level language source code into actions that are immediately performed
- Converts and executes one statement at a time
- Conversion and execution alternate
- Example interpreted languages: Perl, HTML, SAS, Mathematica, Unix “shell” (interactive system within Unix)



# Assembler

---

- Converts a human-readable CPU-specific **assembly code** into CPU-specific, non-human-readable **machine language**
- Like a compiler, but for a low level assembly language instead of for a high level language



# Our Old Friend `hello_world.c`

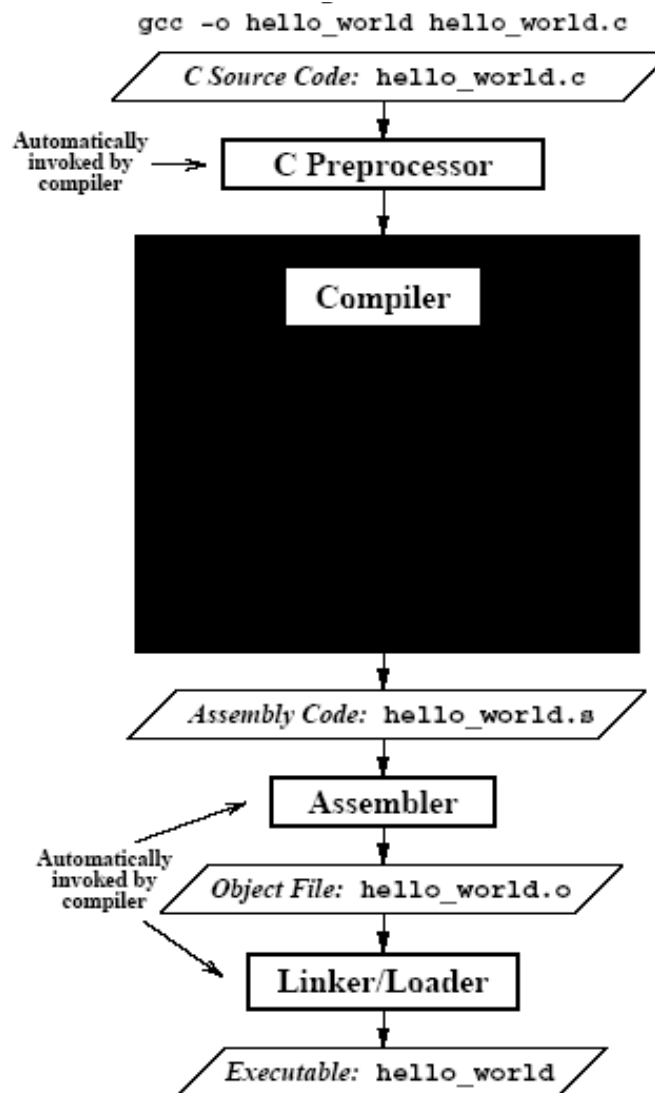
---

```
% cat hello_world.c
/*
*****
*** Program: hello_world ***
*** Author: Henry Neeman (hneeman@ou.edu) ***
*** Course: CS 1313 010 Spring 2025 ***
*** Lab: Sec 012 Fridays 1:00pm ***
*** Description: Prints the sentence ***
*** "Hello, world!" to standard output. ***
*****
*/
#include <stdio.h>

int main ()
{ /* main */
    /*
    *****
    *** Execution Section (body) ***
    *****
    *
    * Print the sentence to standard output
    * (i.e., to the terminal screen).
    */
    printf("Hello, world!\n");
} /* main */
% gcc -o hello_world hello_world.c
% hello_world
Hello, world!
```

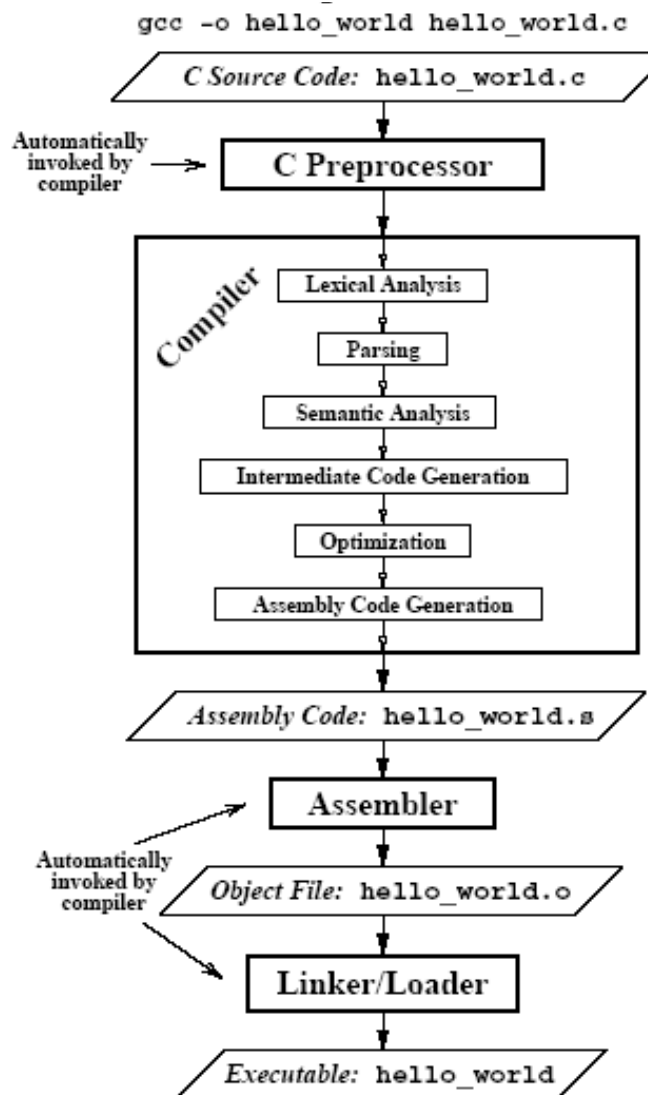


# Compiler Details





# Compiler Details (cont'd)



# Elements of a Compiler #1

---

- **Lexical Analyzer**: identifies a program's “word” elements:
  - **Keywords** (for example, `int`, `while`)
    - These are built into the programming language and cannot be changed by programmers.
  - **Literal constants** (for example, `5`, `0.725`, `"Hello, world!"`, delimited by double quotes on both ends)
  - User-defined **identifiers** (for example, `addend`)
  - **Operators**; for example:
    - Arithmetic: `+` `-` `*` `/` `%`
    - Relational: `==` `!=` `<` `<=` `>` `>=`
    - Logical: `&&` `||` `!`

These will be explained soon.



# Elements of a Compiler #2

---

- **Parser**: determines the program's grammar
- **Semantic Analyzer**: determines what the program does
- **Intermediate Code Generator**: expresses, as an assembly-like program, what the program does
- **Optimizer**: makes code more efficient (faster)
- **Assembly Code Generator**: produces the final assembly code that represents what the program does



# Phases of Compiling

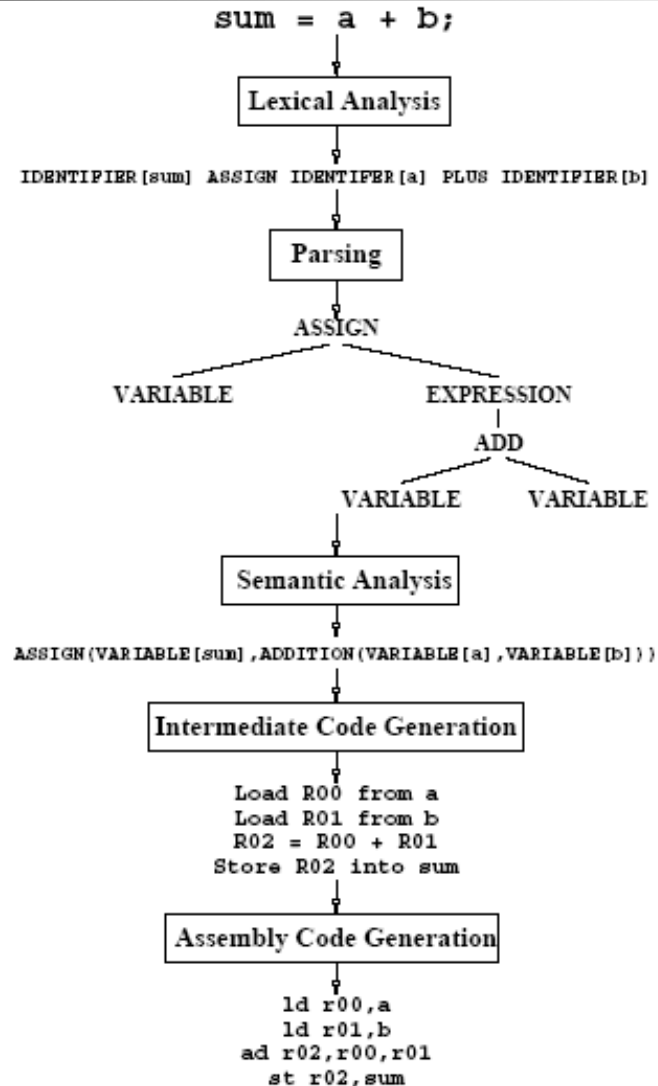
---

- Compiler
- Assembler: turns assembly code into machine code
- Linker/loader: turns machine code into an executable file

Both the assembler and the linker/loader are invoked automatically by the compiler, so you don't have to worry about them.



# Compiling a C Statement



# Assembly Code for `hello_world.c` #1

---

On Pentium4 Using `gcc`

```
pushl %ebp
movl %esp, %ebp
subl $8, %esp
subl $12, %esp
pushl $.LC0
call printf
addl $16, %esp
leave
ret
```

Different *opcodes*  
(assembly language  
instructions)

On IBM POWER4 Using `gcc`

```
mflr 0
stw 31,-4(1)
stw 0,8(1)
stwu 1,-64(1)
mr 31,1
lwz 3,LC..1(2)
bl .printf
nop
lwz 1,0(1)
lwz 0,8(1)
mtlr 0
lwz 31,-4(1)
blr
```



# Assembly Code for `hello_world.c` #2

---

On Pentium4 Using `gcc`

(GNU compiler)

```
pushl %ebp
movl %esp, %ebp
subl $8, %esp
subl $12, %esp
pushl $.LC0
call printf
addl $16, %esp
leave
ret
```

Different sequences of  
instructions  
(using the same  
opcodes)

On Pentium4 Using `icc`

(Intel compiler)

```
pushl %ebp
movl %esp, %ebp
subl $3, %esp
andl $-8, %esp
addl $4, %esp
push $__STRING.0
call printf
xorl %eax, %eax
popl %ecx
movl %ebp, %esp
popl %ebp
ret
```





# Machine Code for `hello_world.c`

---

```
10111101010100010101011110101001
10111010101000010101101011101000
01110101010000101011010111010001
010101001010101011010101011010
```

...



# How to Program in Machine Language Directly

---

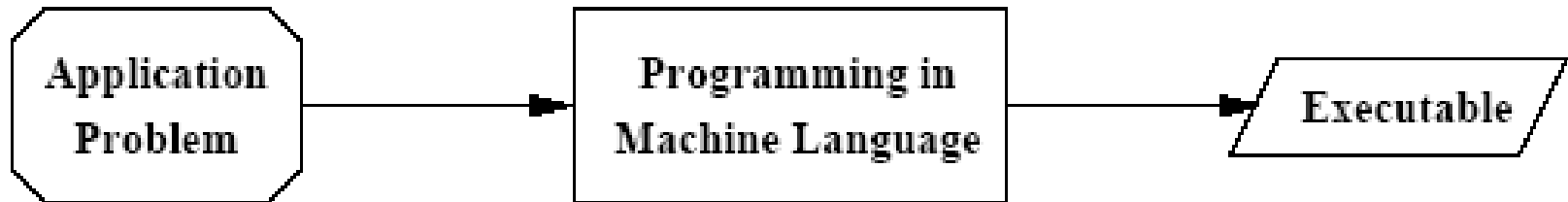
1. Write the assembly code for the program directly by hand on paper (that is, not in a high level language).
2. For each assembly language instruction, look up the bit pattern of the associated machine code.
3. On the computer console, flip switches to match the bit pattern of the machine code.
4. Press the “Run” button.

On modern computers, programming directly in machine language is just about impossible.

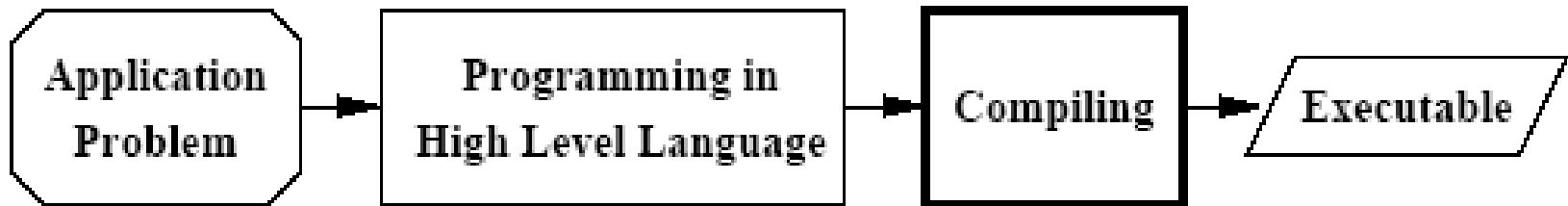


# Why Not Do Everything in Machine Language?

---



Incredibly tedious and ridiculously error-prone!

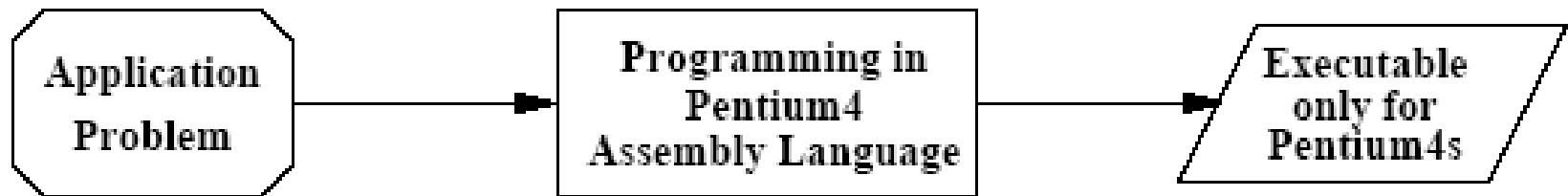


~~Fun and easy!~~

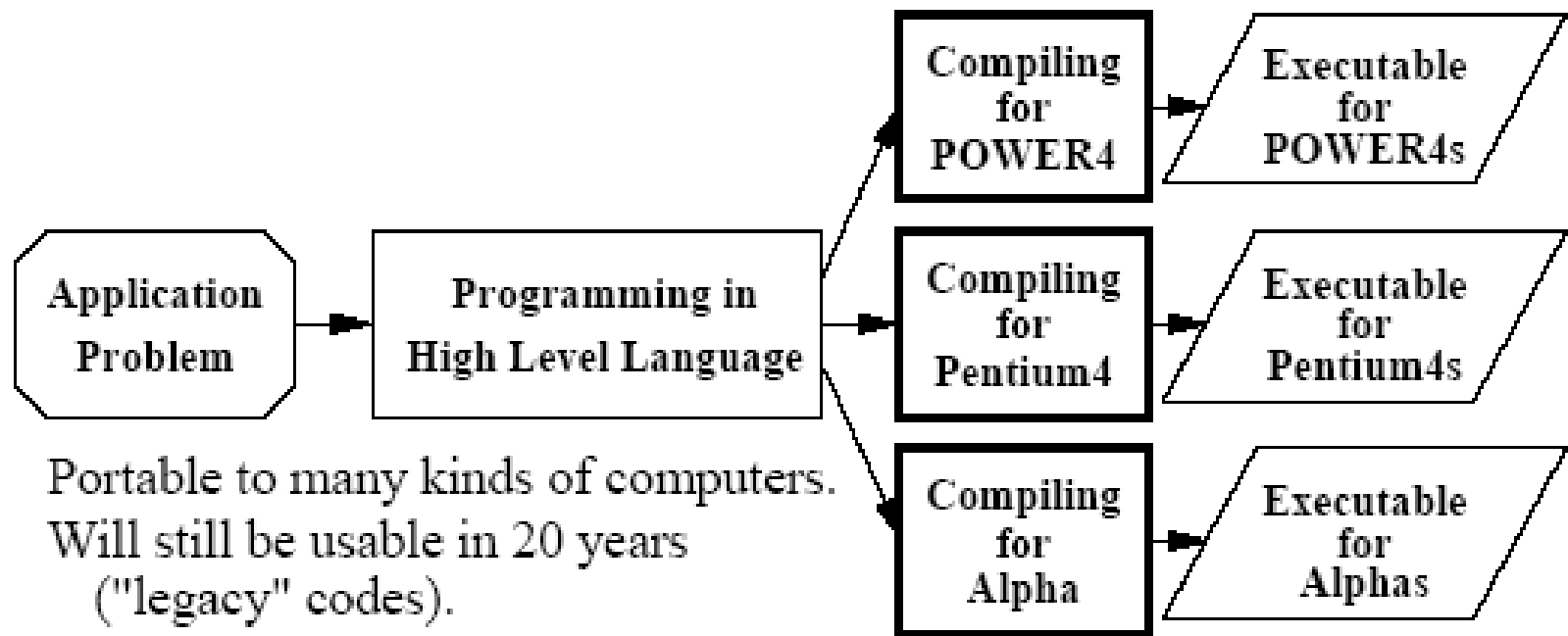
Not nearly as tedious or error-prone!



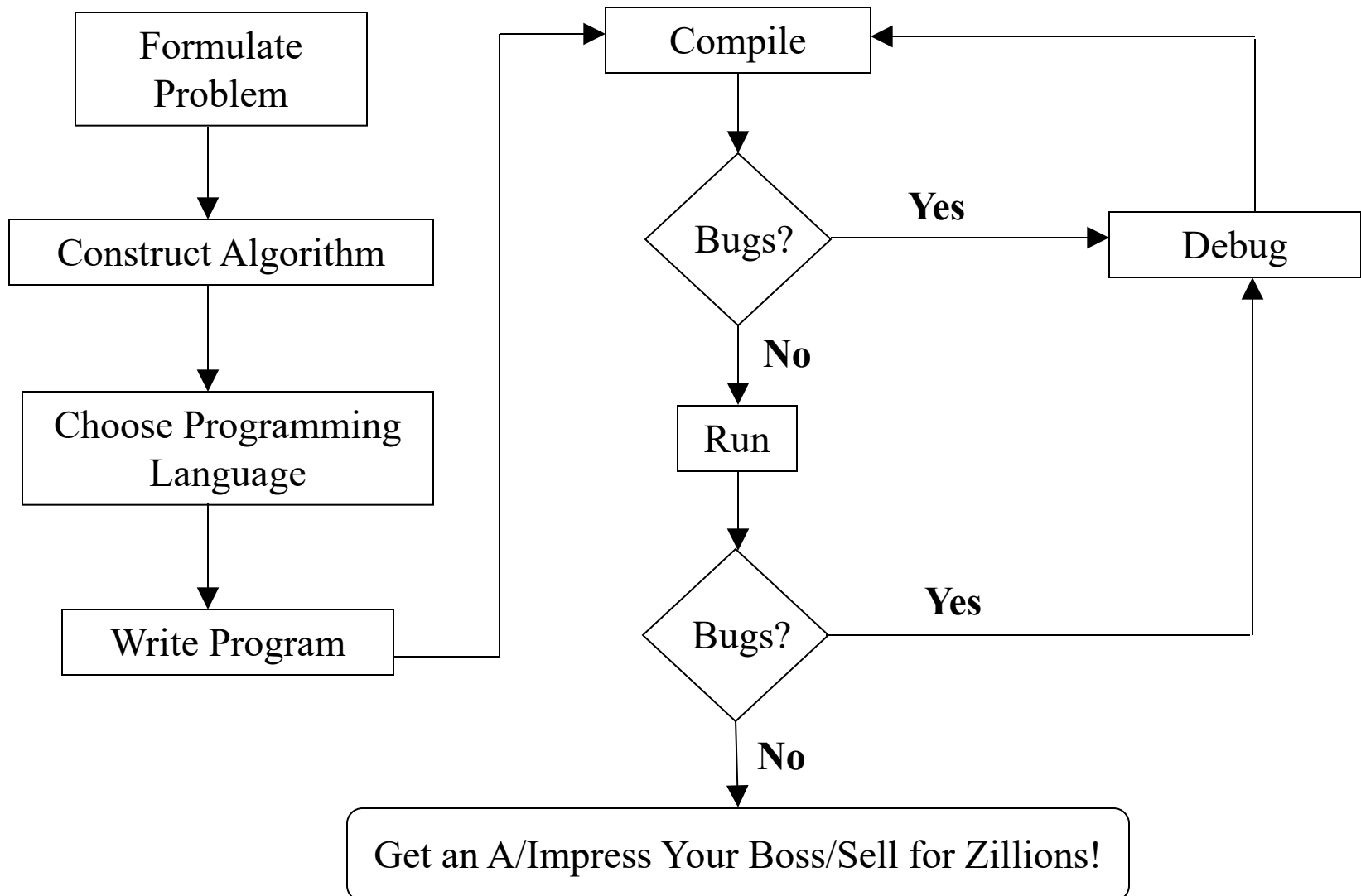
# Why Not Do Everything in Assembly Language?



Can't be run on any other kind of computer.  
May be completely obsolete in a few years.



# The Programming Process



# What is an Algorithm?

---

An algorithm is:

- a step-by-step method
- that is written in a natural language (for example, English) or in pseudocode (something that sort of looks like a programming language but isn't as precise), instead of in a programming language,
- that solves a well-defined (though not necessarily useful) problem,
- on a well-defined set of inputs (which may be empty),
- using finite resources (for example, computing time and storage),
- and that produces a well-defined set of outputs (which may be empty).



# Algorithms

An **algorithm** is a **language-independent** way of expressing the method of solving a problem; that is, an algorithm could be expressed in two different languages (for example, English and Japanese) and still be the same algorithm.

A **program**, by contrast, is a **language-dependent** implementation of the method of solving a problem; that is, the same set of steps expressed in two different programming languages would be two different programs, even if the two programs accomplished exactly the same result.

Many programs, but not all, implement algorithms.

Programs that don't implement algorithms often implement **heuristics**, which typically are inexact but good enough.

The word “algorithm” comes from the name of the 9<sup>th</sup> century mathematician, Muhammad ibn Musa al-Khwarizmi.

<https://en.wikipedia.org/wiki/Algorithm>





# Algorithm Example: Eating a Bowl of Corn Flakes

---

- Get bowl from cupboard
- Get spoon from drawer
- Get box of corn flakes from pantry
- Get jug of milk from refrigerator
- Place bowl, spoon, corn flakes and milk on table
- Open box of corn flakes
- Pour corn flakes from box into bowl
- Open jug of milk
- Pour milk from jug into bowl
- Close jug of milk
- Go to table
- Pick up spoon
- Repeat until bowl is empty of corn flakes
  - Using spoon, pick up corn flakes and milk from bowl
  - Put spoon with corn flakes and milk into mouth
  - Pull spoon from mouth, leaving corn flakes and milk
  - Repeat ...
    - Chew
  - ... until mouthful is mush
  - Swallow
- Leave mess for housemates to clean up



# Top-Down Design

---

Algorithms for most non-trivial problems tend to be fairly complicated.

As a result, it may be difficult to march from an algorithm's beginning to its end in a straight line, because there may be too many details to keep in your head all at one time.

Instead, you can use a technique called **top-down design**: start with the whole problem, then break it into a few pieces, then break each of those pieces into a few pieces, then break each of those pieces into a few pieces, and so on, until each piece is pretty small.



# Eating Cornflakes: Top Level

---

- Get stuff
- Transport stuff
- Set up stuff
- Eat
- Finish

