

# Software Lesson 2 Outline

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

---

- There are said to be over 7000 natural languages in the world:  
<https://www.ethnologue.com/guides/how-many-languages>
- Examples: English, Chinese, Swahili, Navajo, Quechua, Maori
- 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 #2

---

- **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?”**



# Natural Languages #3

---

- **Can be ambiguous:**

**“When did he say she was going?”**

could be interpreted as:

- State the time at which he asked, “She was going?”
- According to him, at what time was she going?



# Natural Languages #4

---

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



# 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 web applets and embedded systems
  - 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 (PC)
  - ARM (handhelds such as cell phones and tablets)
  - IBM POWER (big IBM machines)
- So, for example, a program in x86 assembly language cannot be directly run on a POWER or ARM machine.
- 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 completed before executing the program
- Examples: Fortran 90, 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
- Examples: 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



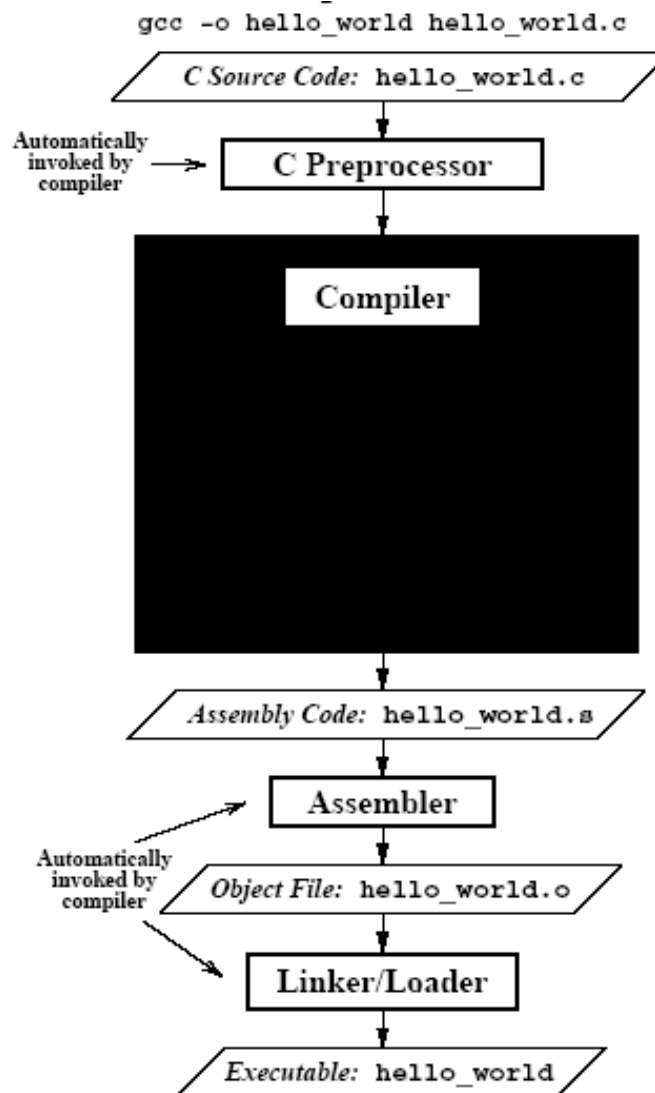
# Our Old Friend `hello_world.c`

```
% cat hello_world.c
/*
*****
*** Program: hello_world ***
*** Author: Henry Neeman (hneeman@ou.edu) ***
*** Course: CS 1313 010 Fall 2017 ***
*** Lab: Sec 014 Fridays 2:30pm ***
*** Description: Prints the sentence ***
*** "Hello, world!" to standard output. ***
*****
*/
#include <stdio.h>

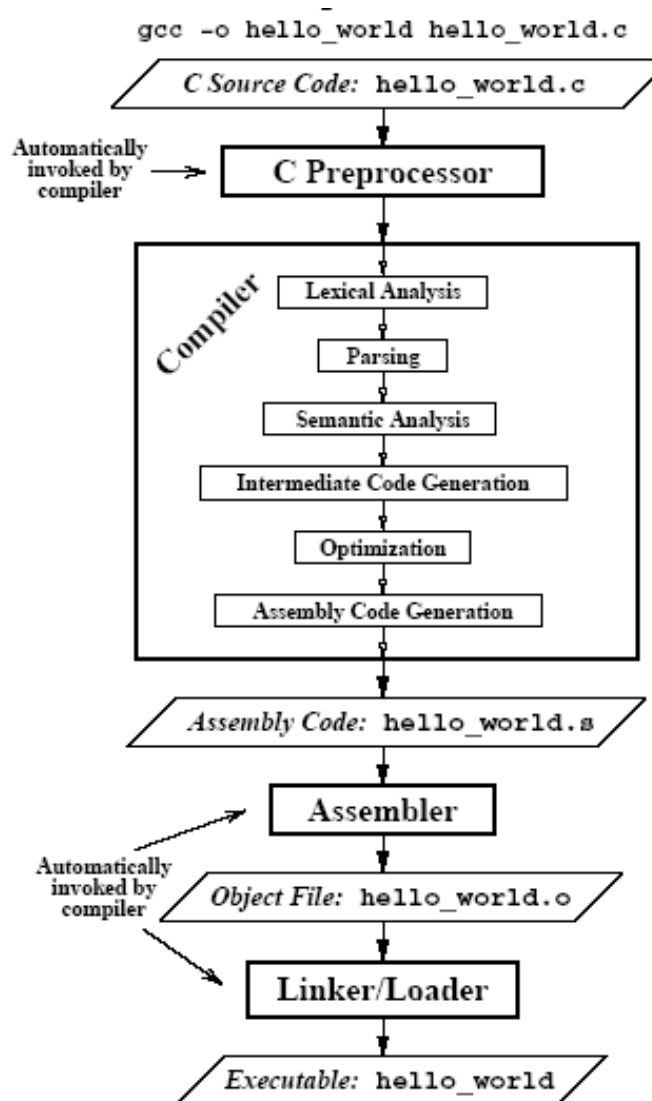
int main ()
{ /* main */
    /*
    *****
    *** Execution Section (body) ***
    *****
    *
    * Print the sentence to standard output
    * (i.e., to the terminal screen).
    */
    printf("Hello, world!\n");
} /* main */
% gcc -o hello_world hello_world.c
% hello_world
Hello, world!
```



# Compiler Details



# Compiler Details (cont'd)



# Elements of a Compiler #1

---

- **Lexical Analyzer**: identifies program's "word" elements
  - Comments (ignored by compiler)
  - **Keywords** (for example, `int`, `while`)
  - **Constants** (for example, `5`, `0.725`, `"Hello, world!"`)
  - User-defined **Identifiers** (for example, `addend`)
  - **Operators**; for example:
    - Arithmetic: `+` `-` `*` `/` `%`
    - Relational: `==` `!=` `<` `<=` `>` `>=`
    - Logical: `&&` `||` `!`



# 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

---

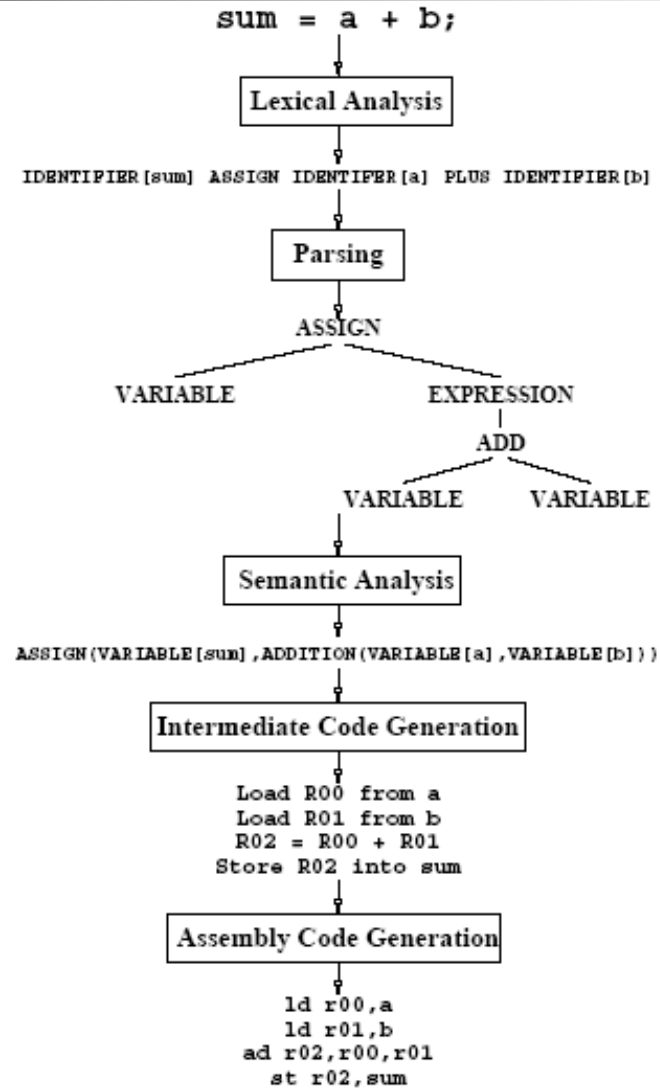
- Compiler
- Assembler: turns assembly code into machine code
- Linker/loader: turns machine code into an executable file

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!*

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!

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

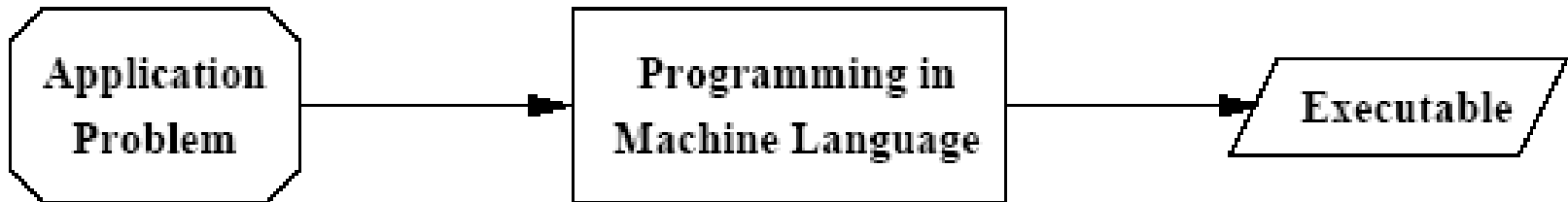
---

1. Write the assembly code for the program directly by hand; 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.

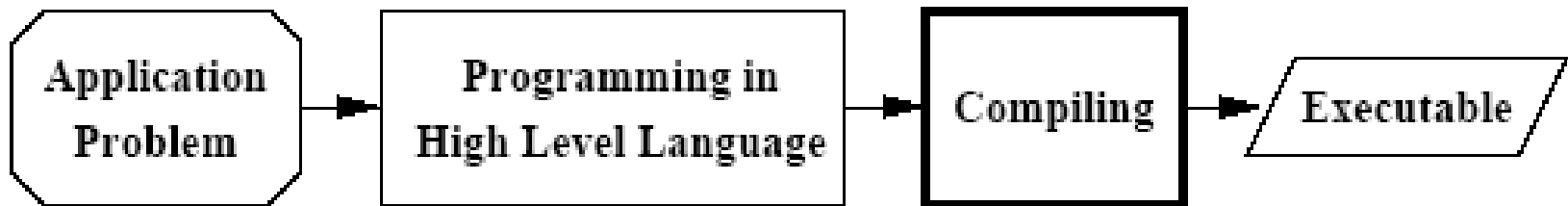
Actually, on modern computers, programming directly in machine language is just about impossible.



# Why Not Do Everything in Machine Language?



Incredibly tedious and ridiculously error-prone!

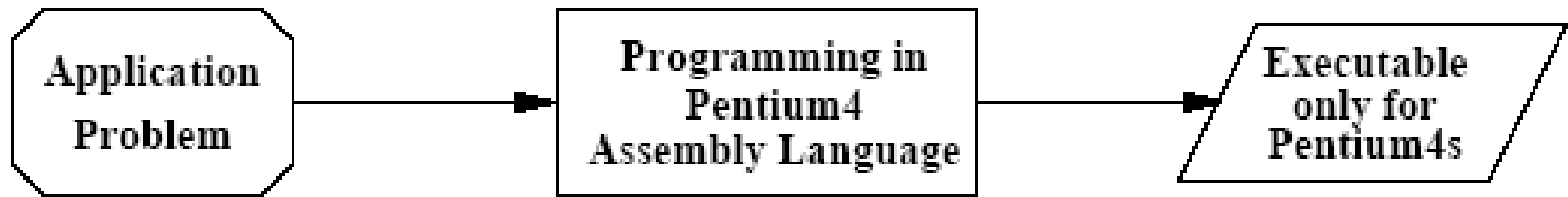


~~Fun and easy!~~

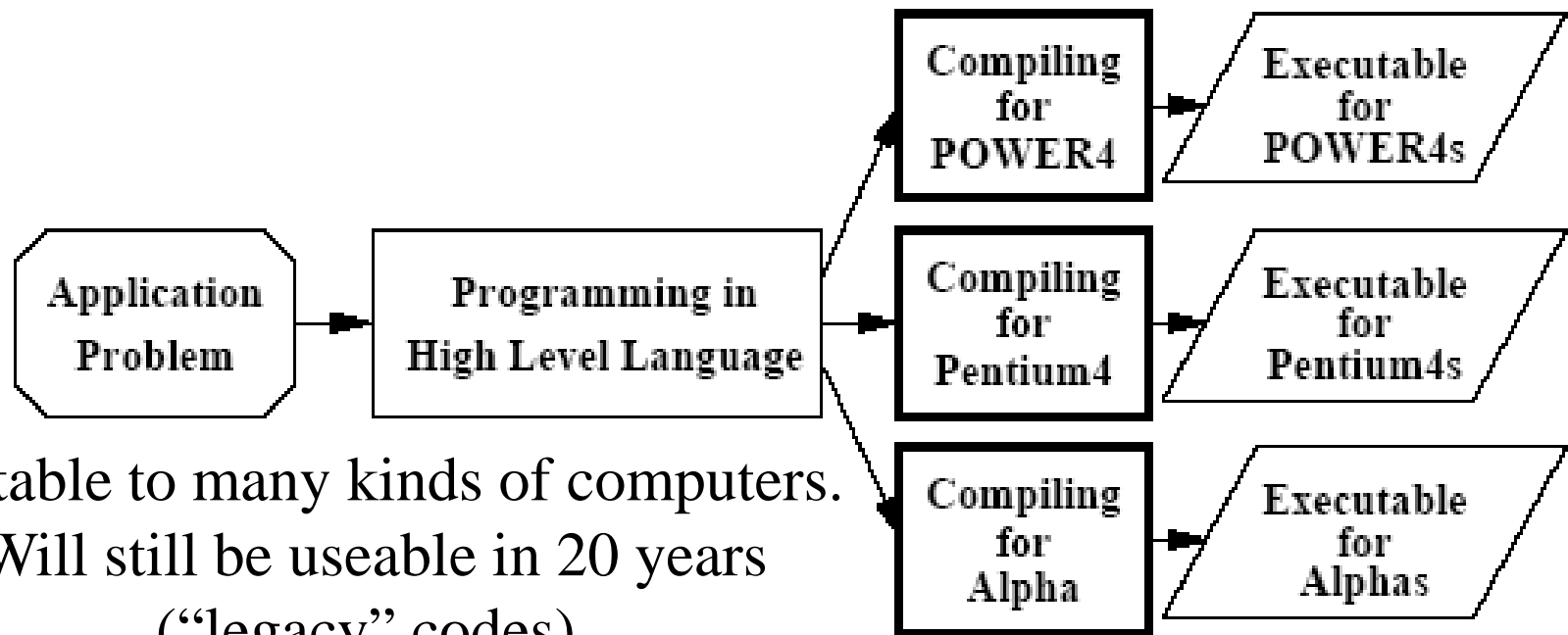
Not nearly as tedious or error-prone!



# Why Not Do Everything in Assembly Language?



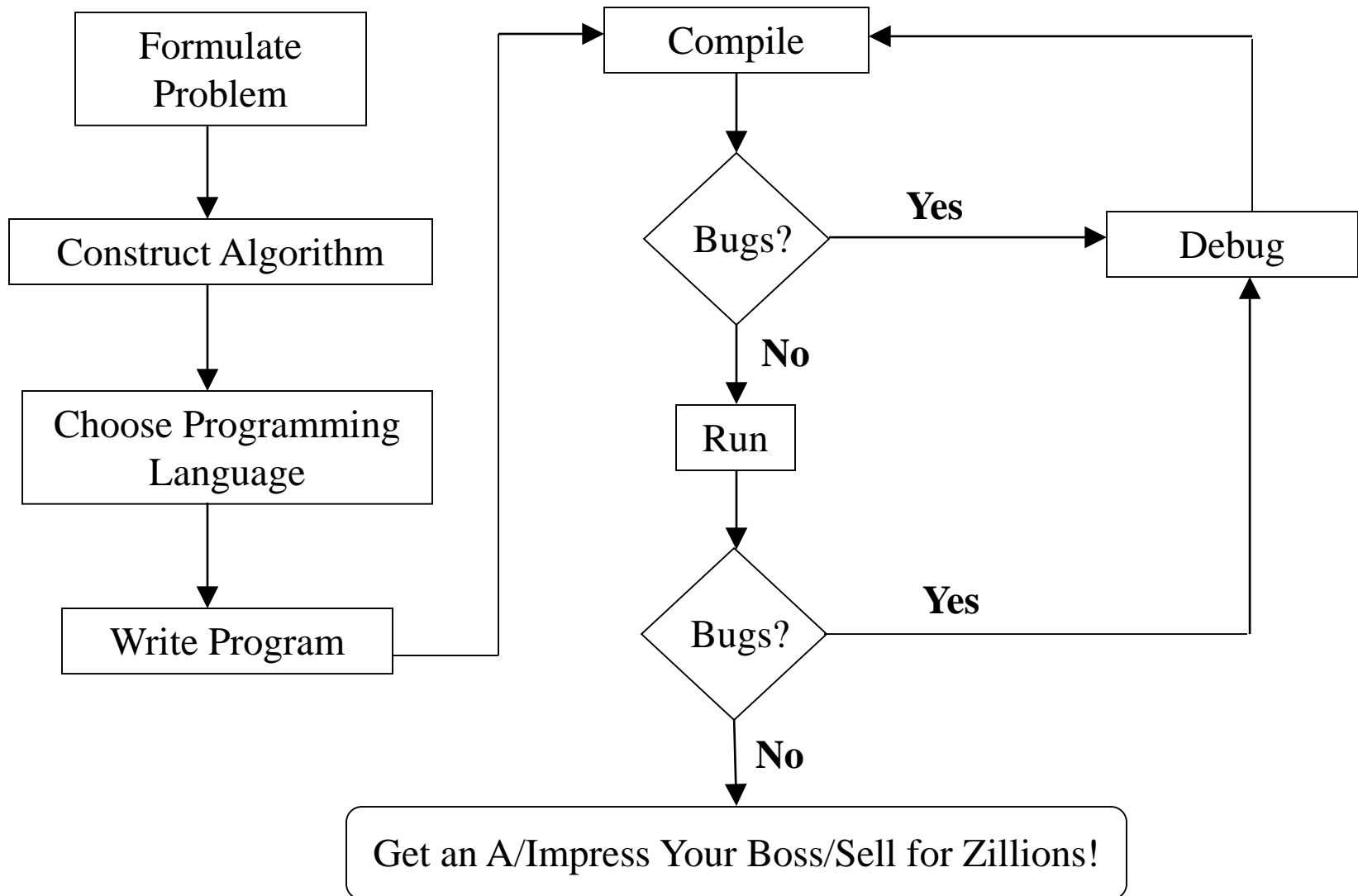
Can't be run on any other kind of computer.  
May be completely obsolete in a few years.



Portable to many kinds of computers.  
Will still be useable in 20 years  
("legacy" codes).



# 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.



# 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

