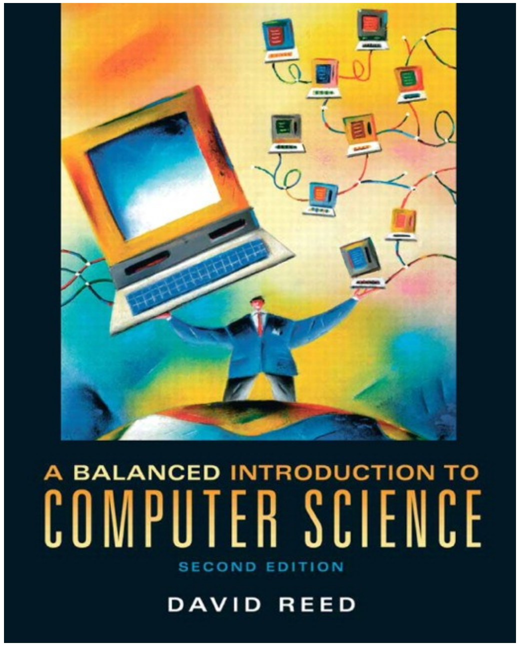
A Balanced Introduction to Computer Science, 2/E

David Reed, Creighton University

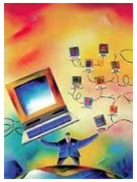
ISBN 978-0-13-601722-6



Chapter 8

Algorithms and Programming Languages

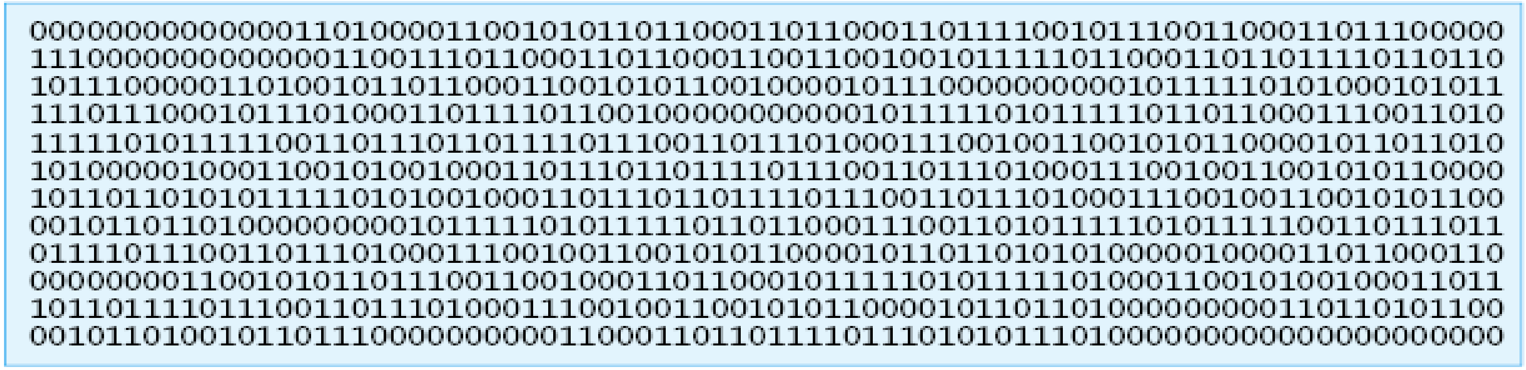
Machine Languages



the first programming languages were known as *machine languages*

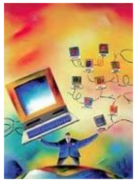
- *a machine language* consists of instructions that correspond directly to the hardware operations of a particular machine
    - i.e., instructions deal directly with the computer’s physical components including main memory, registers, memory cells in CPU
    - very low level of abstraction
- machine language instructions are written in binary
    - programming in machine language is tedious and error prone
    - code is nearly impossible to understand and debug

excerpt from a machine language program:



```
0000000000000001101000011001010110110001101100011011110010111001100011011100000
1110000000000000011001110110001101100011001100100101111101100011011011110110110
1011100000110100101101100011001010110010000101110000000000010111110101000101011
1110111000101110100011011110110010000000000001011111010111110110110001110011010
1111101011111001101110110111101110011011101000111001001100101011000010110110 10
1010000001000110010100100011011101101111011100110111010001110010011001010110000
1011011010101111101010010001101110110111101110011011101000111001001100101011 00
0010110110100000000010111110101111110110110001110011010111110101111100110111011
0111101110011011101000111001001100101011000010110110101010000010000110110001 10
0000000011001010110111001100100011011000101111101011111010001100101001000110 11
1011011110111001101110100011100100110010101100001011011010000000001101101011 00
0010110100101101110000000000011000110110111101110101011101000000000000000000000
```

High-Level Languages



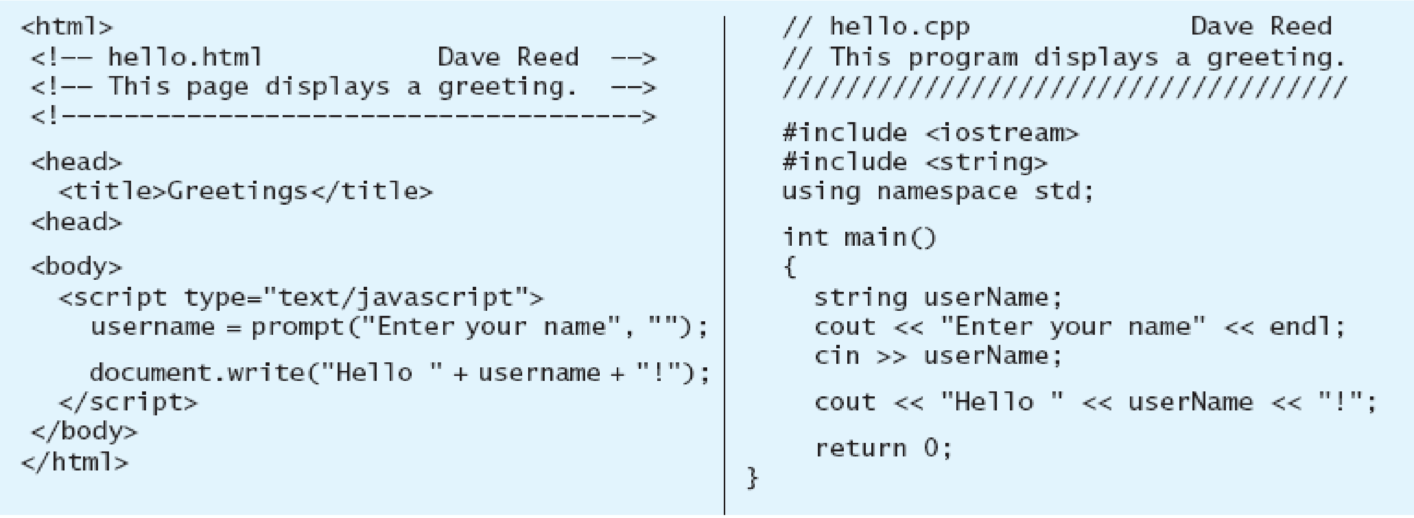
in the early 1950’s, *assembly languages* evolved from machine languages

- an assembly language substitutes words for binary codes
- much easier to remember and use words, but still a low level of abstraction (instructions correspond to hardware operations)

in the late 1950's, *high-level languages* were introduced

- high-level languages allow the programmer to write code closer to the way humans think (as opposed to mimicking hardware operations)
- a much more natural way to solve problems
- plus, programs are machine independent

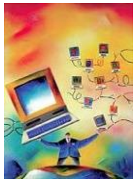
two high level languages that perform the same task (in JavaScript and C++)



```
<html>
 <!-- hello.html           Dave Reed  -->
 <!-- This page displays a greeting.  -->
 <!---------------------------------->

 <head>
   <title>Greetings</title>
 <head>

 <body>
   <script type="text/javascript">
     username = prompt("Enter your name", "");

     document.write("Hello " + username + "!");
   </script>
 </body>
</html>
```

```
  // hello.cpp                Dave Reed
  // This program displays a greeting.
  //////////////////////////////////////

  #include <iostream>
  #include <string>
  using namespace std;

  int main()
  {
    string userName;
    cout << "Enter your name" << endl;
    cin >> userName;

    cout << "Hello " << userName << "!";

    return 0;
}
```

Program Translation



using a high-level language, the programmer is able to reason at a high-level of abstraction

- but programs must still be translated into machine language that the computer hardware can understand/execute

there are two standard approaches to program translation

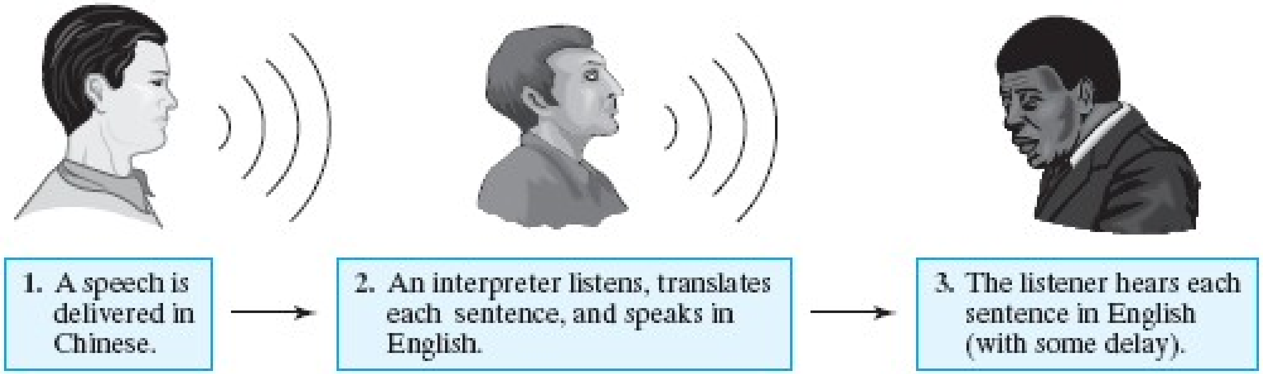
- interpretation
- compilation

real-world analogy: translating a speech from one language to another

- an *interpreter* can be used provide a real-time translation
  - the interpreter hears a phrase, translates, and immediately speaks the translation
  - ADVANTAGE: the translation is immediate
  - DISADVANTAGE: if you want to hear the speech again, must interpret all over again
- a *translator* (or *compiler*) translates the entire speech offline
  - the translator takes a copy of the speech, returns when the entire speech is translated
  - ADVANTAGE: once translated, it can be read over and over very quickly
  - DISADVANTAGE: must wait for the entire speech to be translated

Speech Translation

Interpreter:

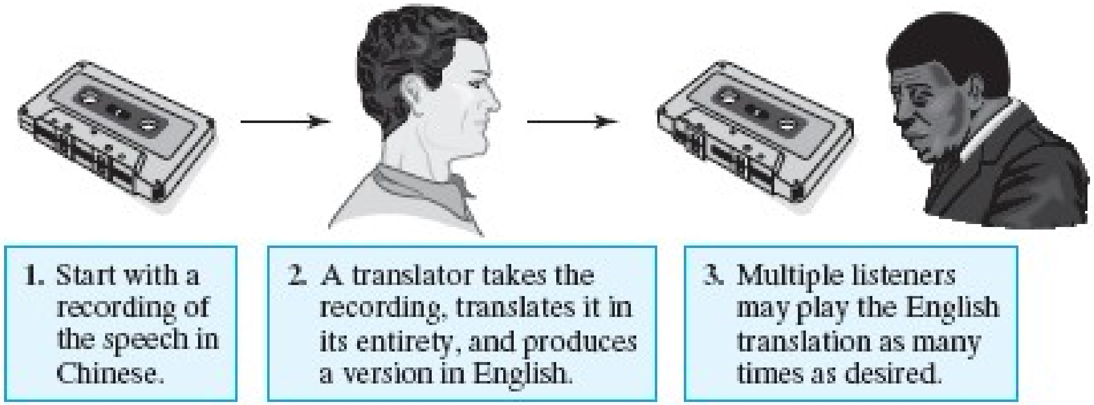


1. A speech is delivered in Chinese.

2. An interpreter listens, translates each sentence, and speaks in English.

3. The listener hears each sentence in English (with some delay).

Translator (compiler):

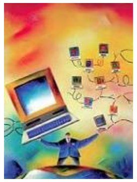


1. Start with a recording of the speech in Chinese.

2. A translator takes the recording, translates it in its entirety, and produces a version in English.

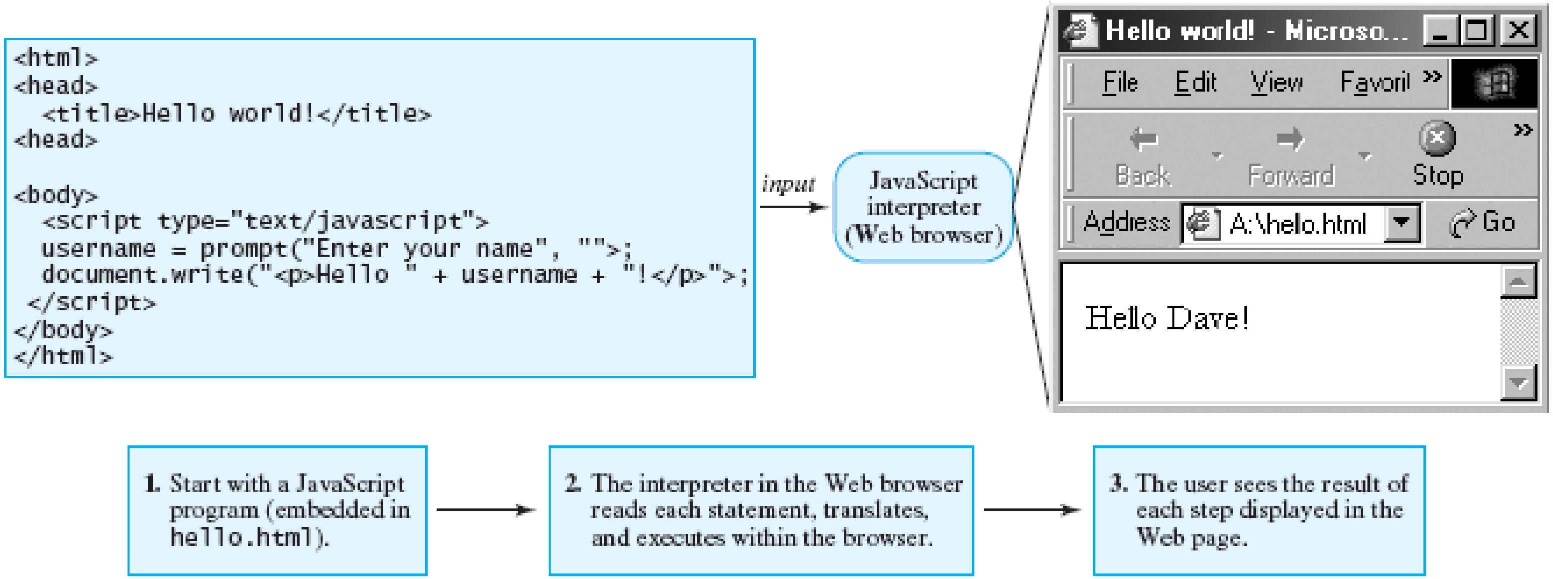
3. Multiple listeners may play the English translation as many times as desired.

Interpreters



for program translation, the interpretation approach relies on a program known as an *interpreter* to translate and execute high-level statements

- the interpreter reads one high-level statement at a time, immediately translating and executing the statement before processing the next one
- JavaScript is an interpreted language



```
<html>
<head>
  <title>Hello world!</title>
<head>

<body>
  <script type="text/javascript">
  username = prompt("Enter your name", "");
  document.write("<p>Hello " + username + "!</p>");
 </script>
</body>
</html>
```

Hello Dave!

1. Start with a JavaScript program (embedded in `hello.html`).

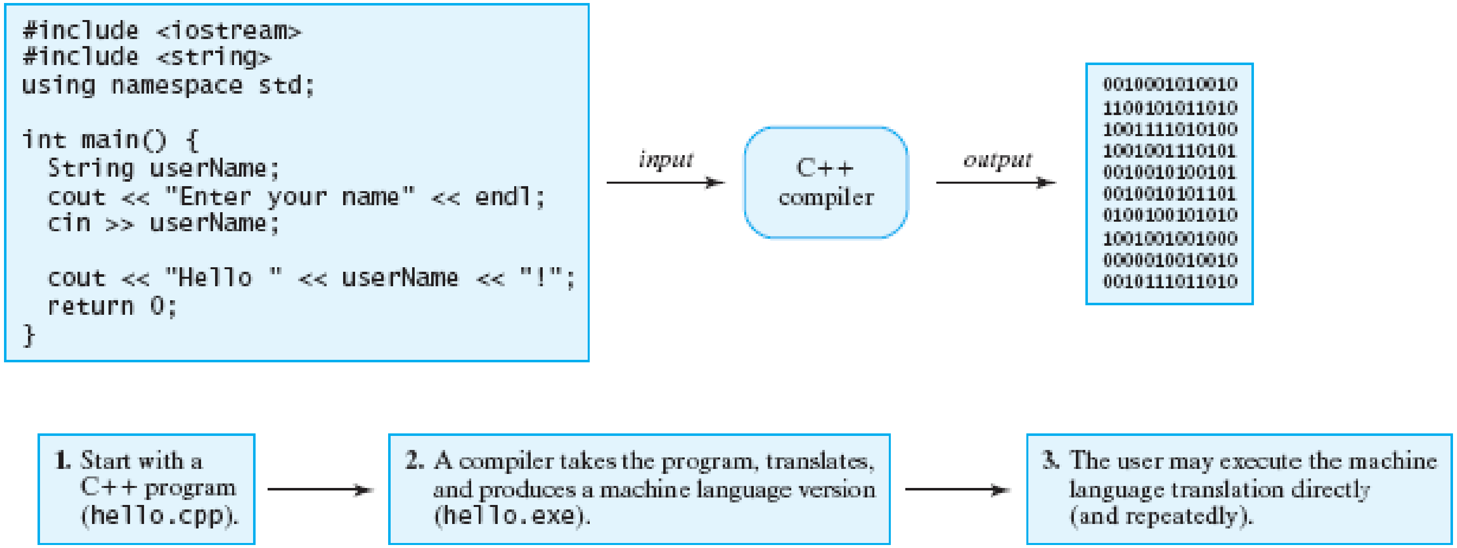
2. The interpreter in the Web browser reads each statement, translates, and executes within the browser.

3. The user sees the result of each step displayed in the Web page.

Compilers

the compilation approach relies on a program known as a *compiler* to translate the entire high-level language program into its equivalent machine-language instructions

- the resulting machine-language program can be executed directly on the computer
- most languages used for the development of commercial software employ the compilation technique (C, C++)

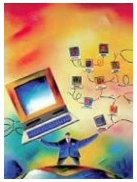


```
#include <iostream>
#include <string>
using namespace std;

int main() {
  String userName;
  cout << "Enter your name" << endl;
  cin >> userName;

  cout << "Hello " << userName << "!";
  return 0;
}
```

```
0010001010010
1100101011010
1001111010100
1001001110101
0010010100101
0010010101101
0100100101010
1001001001000
0000010010010
0010111011010
```

1. Start with a C++ program (hello.cpp).
2. A compiler takes the program, translates, and produces a machine language version (hello.exe).
3. The user may execute the machine language translation directly (and repeatedly).

Interpreters and Compilers



tradeoffs between interpretation and compilation

interpreter

- produces results almost immediately
- particularly useful for dynamic, interactive features of web pages
- program executes more slowly (slight delay between the execution of statements)

compiler

- produces machine-language program that can run directly on the underlying hardware
- program runs very fast after compilation
- must compile the entire program before execution
- used in large software applications when speed is of the utmost importance