Historical evolution of programming languages
Homework #3

- Canceling this homework assignment
- More details on the new HW#3 assignment early next week
- Will update the syllabus to reflect this change soon
How do stored instructions work?

- At the hardware level, a computer can only understand instructions that are encoded in binary (1s and 0s)
- So, as an example, let’s look at how to add two numbers together using the 6502 microprocessor
  - Used in the Apple II, Atari 2600, Commodore 64, and other early microcomputers

- The 6502 has a small number of built-in variables, called registers
- One of these is called the **accumulator**, since it is commonly used to accumulate (hold) intermediate results of computations
- Adding two numbers, say, 7 plus 2, requires:
  - Loading the first number into the accumulator
    - Accumulator = 7
  - Adding the second number to the current value of the accumulator
    - Accumulator = current_value_of_accumulator + 2

6502 Microprocessor
Adding two numbers on the 6502

- Load the accumulator with the value 7
  - In **machine code**, this is:
    - A9 07 (hex) = 10101001 00000111
- Add to the accumulator the value 2
  - 69 02 (hex) = 01101001 00000010
- After executing these two instructions, the accumulator now holds the value of 9 (00001001)

**Machine code**

- A binary representation (encoding) of instructions for a specific microprocessor (or computing engine)

**The 6502 microprocessor contains electronic circuits that understand this binary encoding**

  - Following the fetch-execute cycle, a memory location is loaded
  - The 6502 decodes the 1s and 0s that come back
  - After decoding, the 6502 performs the specified operation, possibly loading additional data from memory
The problem with machine code

- It is, in fact, possible for humans to program a computer (i.e., to describe algorithms for a computer) using just machine code
  - But, it is **terribly slow**, and **very error prone**
  - A one-bit error would make the program perform a different function
  - On some early microcomputers (Altair 8800), this was the only way to program the machine as it came out of the box
    - Other ways were developed quickly, but required some form of storage, like a tape recorder
- A faster way of programming a computer was needed
Solution 1: Assembly language

- The idea behind assembly language is to make it easier for humans to write machine code
  - For each different instruction, have an associated short, human-meaningful acronym or name
  - These short acronyms or names are often called **mnemonics**, and all of them together are known as an **assembly language**
    - So, for LoaDing the Accumulator, the short form is “LDA”
    - For adding a number to the accumulator, the short form is “ADC”
      - Stands for ADd with Carry
  - A program, called an **assembler**, takes assembly language as input, and produces machine code as output
Let’s revisit adding two numbers together on the 6502

- Load the accumulator with the number 7
  - LDA #$07
  - Load the accumulator with the constant value of hexadecimal 7 = decimal 7
- Add to the current accumulator value the number 2
  - ADC #$02
  - Add the constant value of hexadecimal 2 = decimal 2 to the accumulator

A 6502 assembler takes these assembly language instructions and converts them into machine code

- LDA #$07 (assembly) → 10101001 00000111 (binary)
- ADC #$02 (assembly) → 01101001 00000010 (binary)

Humans can more easily understand the assembly language, while the 6502 hardware can understand the machine code
History of assembly language

- Assembly languages appear to have emerged in the 1950s
- An IBM users group, called SHARE, was formed in 1955
- SHARE members were instrumental in contributing to an assembler for the IBM 704 machine
- However, there might have been earlier examples
  - Even doing a little bit of machine code programming would make you yearn for a way of automating it
The problem with assembly language

- Assembly language has several problems
  - While faster than machine code, it is still **slow to write**
  - It is **hard to understand** another person’s assembly language program
    - Hence, large assembly language programs are **hard to maintain**
  - It is challenging to maintain and use large collections of pre-made assembly language algorithms (functions, procedures)
    - **Macros**, assembly language codes that expand to multiple basic operations helped, but were insufficient
    - So, assembly language programs are **hard to scale**, and **don’t support reuse**
  - Writing an algorithm in assembly language requires programmers to break it down into **very** elementary steps
    - For example, the 6502 doesn’t have floating point mathematics built into the microprocessor
    - So, in assembly language, you need to write (or reuse) a routine for adding, multiplying, dividing, etc. floating point numbers
      - See: [http://www.6502.org/source/floa](http://www.6502.org/source/floa)ts/wozfpl.txt for an example
    - This is called programming at a **low level of abstraction**
Benefits of assembly language

- **Zowie, it is fast**
  - A programmer who deeply understands the microprocessor and the computer architecture can write very fast and efficient code
  - This can be important for sections of an algorithm that are executed very frequently, and hence small speed improvements have a big impact

- **Compact**
  - Assembly language code typically uses memory in an efficient way
  - For this reason, it is sometimes used for embedded computing, where the available memory is low

- **Hardware drivers**
  - Since it is so low-level, assembly can be useful for interacting directly with hardware devices connected to a computer (since this hardware is also very low-level)
Programming languages

- The underlying idea of programming languages is:
  - To make it possible to write algorithms in a human-friendly notation
  - To have this notation allow thinking at a higher level of abstraction than assembly/machine language supports

- So, for example
  - In a programming language, to add 2 + 7, we might write:
    - `sum = 2 + 7`
    - The variable named “sum” takes the value of 2 plus 7
    - Much easier to understand than LDA/ADC
    - Allows the desired logical operation (i.e., the addition) to be directly expressed, instead of broken down into two separate operations
Compilation

- How does a program written in a programming language turn into machine code?
- There are several techniques, this slide discusses **Compilation**
  - As opposed to interpreted languages (more on this later)

- **Process:**
  - A programmer writes the source code of a program
    - Source code: A file of text with programming language instructions
  - A program called a compiler converts this text into assembly language
  - An assembler then converts the assembly language into machine code

Microprocessor hardware can understand and execute
Grace Hopper

- PhD mathematics, Yale, 1934
  - Part of the post-WWI generation of women mathematicians
- Associate Professor at Vassar, joined US Navy in 1943
- Among the first programmers of the Harvard Mark I computer
  - And hence, among the first modern computer programmers
- Developed the first compiler
- Developed the idea of machine independent programming languages
  - This work led to the development of the language COBOL
- Credited with popularizing the term “debugging” to get errors out of a program
- Today, the Grace Hopper Celebration of Women in Computing is a yearly conference focused on the contributions of women in computing.
- Watch: 60 Minutes interview with Hopper
  - http://www.youtube.com/watch?v=7sUT7gFQEsY
In mid 1950’s (1953-1957), John W. Backus (IBM) led a team that developed the language FORTRAN

- IBM Mathematical Formula Translating System
- Developed at IBM campus in south San Jose, California

As befits the name, is well suited for numeric computation and scientific computing

Though not considered a “hip” language, is still in widespread use today, and has been continuously updated

- Most recent update was in 2008, which included features for parallel execution
FORTRAN (cont’d)

- Major factors in its adoption:
  - Speed: had to overcome bias of assembly language programmers that compiled code is much slower than assembly language code
    - As a result, first FORTRAN compiler focused on code optimization
    - That is, ensuring that the assembly language code generated by the compiler runs quickly
  - Efficiency: FORTRAN code typically has 1 statement for every 20 statements in an assembly language program
    - Programmers could write FORTRAN code more quickly than assembly
  - Portability: FORTRAN code written on one computer could be executed on a different type of computer, with different machine code

- John Backus, in a 1979 interview in Think, IBM Employee magazine:
  - “Much of my work has come from being lazy. I didn’t like writing programs, and so, when I was working on the IBM 701, writing programs for computing missile trajectories, I started work on a programming system to make it easier to write programs.”
The two other influential early programming languages from the 1950s were LISP and COBOL.

**LISP (1958)**
- LISt Processor, invented by John McCarthy
- Commonly used in artificial intelligence programming, but has widespread utility
- Variants in use today: Scheme, Common Lisp

**COBOL (1959)**
- COnmon Business-Oriented Language
- Widely used for business information technology applications
- In declining use today, but still used, and the language was updated in 2002