Today

- Syllabus overview
- HW1 - due next Wednesday
- Intro to the topic
First Question:

What programming languages have you used?

- Python
- C++ / C
- Java
- Scala
- PHP
- Perl
- Assembly
- C#
- JavaScript
- Matlab
- VHDL
- Visual Basic
- Swift
Categories

High-level versus low-level:

\[ \text{Python} \quad \text{C} \]

\[ \text{Assembly} \xrightarrow{\text{assembler}} \text{Machine code} \]

\[ \text{High-level} \xrightarrow{\text{compiler}} \text{Machine or assembly} \]
High-Level Languages

- Began in 1950's with Fortran
- First machine-independent solution
- Slow to become popular because compileless were not as good as humans

(Not true now—plus, labor costs more than hardware!)
Why so many?

- Evolution: Still very new!
  - Structured programming (using loops instead of go-to) was only developed in the late 60's.
  - Object orientation was developed in the 80's.

- Personal preference
- Special purposes: Often, the choice depends on what you want to do!
  - C is good for low level systems work
  - Prolog is good for logical relationships among data
  - Awk is good for character and string manipulation
  - Python and Perl are good scripting tools
Other issues

- Learning curve
- Ease of use
- Standardization
- Open Source
- Good Compilers
- Economics & patronage
- Inertia
Families of high-level Languages

1. Declarative Languages:
   - Focus is on *what* the computer should do
   - "higher-level"

2. Imperative Languages:
   - Focus is on *how* the computer should do it
   - dominant paradigm - often better performance
Imperative

Categories:

   - based on computation with variables

B. Scripting languages: bash, awk, PHP, Perl, Python, Ruby, etc.
   - subset of von Neumann, but tailored for ease of expression over speed

C. Object-oriented: traced from Simula 67, often related to von Neumann, but object-based
Declarative

Categories & Examples:

A) Functional languages: Lisp, Scheme, ML, **Haskell**
   - based on recursive definition of functions
     (inspired by lambda calculus)

B) Logic-based: Prolog, SQL (?)
   - computation is based on attempts to find values that satisfy specified relationships

C) Data flow: *Id* → *Val*
   - flow of information (tokens) among nodes
Example: Compute the gcd (stolen from my IS0 lecture)

- Reset \( u + v \) to values \( u \) and \( v \), respectively.
- Divide \( u \) by \( v \) and let \( r \) = remainder.
- Set \( u \) = \( v \) equal to the numbers.
- Is \( v = 0 \)?
  - Yes: Output \( u \)
  - No: Go back to divide \( u \) by \( v \), and repeat.
GCD in a functional language

\[ \text{gcd}(a,b) := \begin{cases} a & \text{if } a=b \\ \text{gcd}(b,a-b) & \text{if } a>b \\ \text{gcd}(a,b-a) & \text{if } b>a \end{cases} \]

\[ \text{gcd}(10,5) := \text{gcd}(5,5) := 5 \]
GCD in Prolog

\[ \text{gcd}(a, b, g) \text{ is true if:} \]

- \( a = b = g \)
- \( a > b \) and \( \exists c \) such that \( c = a - b \) and \( \text{gcd}(c, b, g) \) is true
- \( b > a \) and \( \exists c \) s.t. \( c = b - a \) and \( \text{gcd}(c, a, g) \) is true

\[ \text{gcd}(10, x, y) \]

A list of \( x, y \)'s that make it true
Why study this?!  \[ x = y = 5 \]

- Choosing appropriate language is a key step.
- Make learning new languages easier.
- Common terminology for comparison and understanding.
- Understand hidden "features".
- Know actual implementation costs.
Compilation versus interpretation

2 models:

**C++**

Source program

Compiler

Input → Target program → Output

**Python**

Source program

Interpreter

Input → Output
Pros & Cons

Interpreters:
- greater flexibility
- better debugging
- better with data that is dependant on input

Compilation:
- much faster
Compilation vs. Interpretation

In reality, most languages are both.

This is the key.
Compilers

The process by which programming languages are turned into assembly or machine code is important in programming languages.

We'll spend some time on these compilers although it isn't a focus of this U class.