CS 441 - Programming Languages

Today

- Syllabus overview

- HW1 - due next Tuesday

- Intro to the topic

- NO CLASS ON THURSDAY!
  (so plenty of time for your HW)
First Question:
What programming languages have you used?
- Python
- C++
- Java
- Matlab
- Objective C
- C#
- Lua
- C
- Ruby
- PHP
- Go
- Lisp
- JavaScript
- SQL
- HTML
- Assembly
Categories

High-level versus low-level:

assembly $\rightarrow$ <assembler> $\rightarrow$ machine code

high-level $\rightarrow$ <compiler> $\rightarrow$ machine or assembly
High-Level Languages

- Began in 1950's with Fortran
- First machine-independent solution
- Slow to become popular because compilers 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 goto) 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 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
   (Object orientation)
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)

1. **Set** \( u = u' \) **equal to the numbers**
2. **Reset** \( u + v \) to values \( v \) and \( r \), respectively
3. **Divide** \( u \) by \( v \) and let \( r = \text{remainder} \)
4. **Is** \( v = 0 ? \)
   - **Yes**: output \( u \)
   - **No**: Go back to step 3
GCD in a functional language

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

Claim: This is equivalent to previous algorithm.
GCD in Prolog

gcd(a, b, g) is true if:

- a = b = g
- a > b and ∃c such that c = a - b and gcd(c, b, g) is true
- b > a and ∃c s.t. c = b - a and gcd(c, a, g) is true
Why study this?

- Choosing appropriate language is a key step.
- Make learning new languages easier.
- Common terminology for comparison and understanding.
- Understand hidden "features". \[ \text{ex: } \text{if} \,(a = b) \]
- Know actual implementation costs. \[ \text{ex: } \text{if} \,(0) \,\text{of} \,(0) \]
Compilation versus Interpretation
2 models:

C/C++:

Source program → Compiler

Target program → Output

Python:

Source program → Interpreter

Output

Optimized for my computer.
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.

How much does a translator do?
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 class.