CSC 2/454 Programming Language Design and Implementation
==========
Course Introduction

Language Design and Language Implementation go together
    an implementor has to understand the language
    a language designer has to understand implementation issues
    ** a good programmer has to understand both

LOTS of programming languages
    Wikipedia's list has 671 entries as of Aug. 2020
        those are just the "notable" ones

Why are there so many programming languages?
    evolution -- we've learned better ways of doing things over time
    diverse ideas about what is pleasant to use
    orientation toward special purposes (SQL)
    orientation toward special hardware (assembly, CUDA)
    market factors: desire to control, or avoid what others control
        (COBOL, PL/I, Ada, Swift, ...)

What makes a language successful?
    easy to learn (BASIC, Scheme, LOGO, Python)
    "powerful" -- easy to express complicated things (if fluent)
        (C++, Common Lisp, Haskell, Perl, APL)
    easy to implement (BASIC, Forth)
    possible to compile to very good (fast/small) code (C, Fortran)
    exceptionally good at something important (PHP, Ruby on Rails, R, SQL)
    backing of a powerful sponsor (COBOL, Ada, Visual Basic, C#, Swift)
    wide dissemination at minimal cost (Pascal, Java, Python, Ruby)
    market lock-in (Javascript)

Why do we have programming languages? -- what is a language _for_?
    abstraction of virtual machine -- way of specifying what you want
        the hardware to do without getting down into the bits
      * languages from the implementor's point of view
    way of thinking -- way of expressing algorithms
      * languages from the user's point of view

    This course tries to balance coverage of these two angles. We will
    talk about language features for their own sake, and about how they
    can be implemented.

  * Knuth: Computer Programming is the art of explaining to another
    human being what you want the computer to do.

This course should help you
    learn new languages more easily
    pick the right language for the task at hand (given a choice)
    choose among alternative ways to express things in a given language
    understand what a compiler does to your code
        for performance and (sometimes) correctness debugging
    emulate useful features in languages that lack them
    use language & compiler technology in your own projects
        almost every complex system has an input language
    prepare for 2/455 :-)

Key to all of this is understanding the _concepts behind_ language design --
thinking about languages NOT in terms of syntax but in terms of
    naming & binding (early? late?)
    data types and abstraction mechanisms
    control flow
    closures
    concurrency
    ...

Units on
    syntax
    semantics
    functional programming
    names
    scripting
    control flow
    type systems
    concurrency
    composite types
    subroutines
    objects
    run-time systems

    (see the web site)

Traditional to group languages in terms of "paradigm"
    imperative
        von Neumann   (Fortran, Ada, Pascal, Basic, C, ...)
        object-oriented   (Smalltalk, Eiffel, C++, Java, C#,
                                    Swift, OCaml, ...)
        scripting   (perl, Python, PHP, Ruby,
                                    Javascript, Matlab, R, ...)
    declarative
        functional   (Scheme/Lisp, ML/OCaml/Haskell/F#)
        logic, constraint-based (Prolog, OPS5, spreadsheet, XSLT)

Not clear this ever really made sense: categories are not mutually exclusive,
and have been getting less so over time.

Today, probably best to talk about paradigms a language _supports_
rather than "the" paradigm to which it belongs.

We'll discuss all of this much more as the semester goes on. For now:

    Imperative languages emphasize computation by modifying variables.
        This allows you to do unbounded amounts of work in loops.

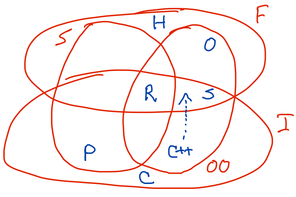
    Functional languages emphasize computation by creating, manipulating,
        and invoking functions. This allows you to do unbounded amounts of
        work via recursion.

    Object oriented languages emphasize structuring the code around
    abstract data types and their operations (methods).

    Scripting languages emphasize delayed decision making and programmer
    flexibility.

    Logic languages emphasize the search for values that satisfy certain
    constraints. We'll touch on them a few times this semester, but
    they won't get as much emphasis as the others (sorry!)

So: paradigms sort of give us a Venn diagram:
    Haskell
    C
    OCaml
    C++
    Ruby
    Scala
    Perl
    eLisp



Imperative languages have historically dominated -- usually, today, with
object oriented features.
    bulk of our attention in this course
BUT
    one unit and lots of scattered attention to functional languages
    lots of functional features making their way into
        mostly-imperative languages -- Scala, Swift, Ruby, Python, ...
            lambda expressions
            functions as arguments and return values
            list comprehensions
            continuations

The imperative and functional paradigms tend to encourage different ways
of thinking about algorithms. I'll be talking about this a lot, and
encouraging you to think in both ways (because neither is better)

Consider insertion sort. In no particular languages:

```
imperative sort(A):
    for i in len(A)-2 downto 0
        v = A[i]
        for j in i+1 to len(A)-1
            // A[j..len(A)-1] is sorted
            if A[j] > v break
            A[j-1] = A[j]
        A[j-1] = v

functional sort(A):
    if len(A) < 2 return A
    else
        let v be A[0] and R be A[1..]
        return insert(v, sort(R))
    where insert(v, S):
        if len(S) == 0 return { v }
        else
            let w be S[0] and T be S[1..]
            if v < w return v . S
            else return w . insert(v, T)
```

These implement the same algorithm. They are likely to compile to
nearly identical machine code. But
(a) The functional version has no assignments.
(b) The imperative version has a more obvious implementation.
(c) when I wrote these in C and Scheme, I had to fix two bugs in the C
    version, but the Scheme one ran the first time.

Will probably draw examples from about 40 languages this semester
Will do projects in 6 or 8 of them
By the time we're done, you should be able to pick up a new language in
    a weekend (though becoming an expert will still take time)

Please watch the lecture on course administration.