Programming

- it's hard to do the programming to get something done
- details are hard to get right, very complicated, finicky
- not enough skilled people to do what is needed
- therefore, enlist machines to do some of the work
  - leads to programming languages

- it's hard to manage the resources of the computer
- hard to control sequences of operations
- in ancient times, high cost of having machine be idle
- therefore, enlist machines to do some of the work
  - leads to operating systems
Evolution of programming languages

- **1940's: machine level**
  - use binary or equivalent notations for actual numeric values

- **1950's: "assembly language"**
  - names for instructions: ADD instead of 0110101, etc.
  - names for locations: assembler keeps track of where things are in memory; translates this more humane language into machine language
  - this is the level used in the "toy" machine
  - needs total rewrite if moved to a different kind of CPU

```
loop  get           # read a number
    ifzero  done  # no more input if number is zero
    add     sum   # add in accumulated sum
    store   sum   # store new value back in sum
    goto    loop  # read another number

done  load    sum   # print sum
     print
     stop

sum   0   # sum will be 0 when program starts
```
Evolution of programming languages, 1960's

- "high level" languages -- Fortran, Cobol, Basic
  - write in a more natural notation, e.g., mathematical formulas
  - a program ("compiler", "translator") converts into assembler
  - potential disadvantage: lower efficiency in use of machine
  - enormous advantages:
    - accessible to much wider population of users
    - portable: same program can be translated for different machines
    - more efficient in programmer time

```
sum = 0
10  read(5,*) num
    if (num .eq. 0) goto 20
    sum = sum + num
    goto 10
20  write(6,*) sum
    stop
    end
```
Evolution of programming languages, 1970's

- "system programming" languages -- C
  - efficient and expressive enough to take on any programming task
    writing assemblers, compilers, operating systems
  - a program ("compiler", "translator") converts into assembler
  - enormous advantages:
    accessible to much wider population of programmers
    portable: same program can be translated for different machines
    faster, cheaper hardware helps make this happen

```c
#include <stdio.h>
main() {
    int num, sum = 0;
    while (scanf("%d", &num) != -1 && num != 0)
        sum += num;
    printf("%d\n", sum);
}
```
C code compiled to assembly language (SPARC)

```c
#include <stdio.h>

main() {
    int num, sum = 0;
    while (scanf("%d", &num) != -1 && num != 0)
        sum = sum + num;
    printf("%d\n", sum);
}

(You are not expected to understand this!)
```

```
.LL2:   add     %fp, -20, %g1
         sethi   %hi(.LLC0), %o5
         or       %o5, %lo(.LLC0), %o0
         mov       %g1, %o1
         call    scanf, 0
         mov       %o0, %g1
         cmp       %g1, -1
         be       .LL3
         ld [%fp-20], %g1
         cmp       %g1, 0
         be       .LL3
         ld [%fp-24], %g1
         ld [%fp-20], %o5
         add       %g1, %o5, %g1
         st       %g1, [%fp-24]
         b        .LL2

.LL3:   sethi   %hi(.LLC1), %g1
         or       %g1, %lo(.LLC1), %o0
         ld [%fp-24], %o1
         call    printf, 0
         mov       %g1, %i0
         ret
```
C code compiled to assembly language (x86)

```c
#include <stdio.h>
main() {
    int num, sum = 0;

    while (scanf("%d", &num) != -1 && num != 0)
        sum = sum + num;
    printf("%d\n", sum);
}
```

```
.L2:    leal    -4(%ebp), %eax
        movl    %eax, 4(%esp)
        movl    $.LC0, (%esp)
        call    scanf
        cmpl    $-1, %eax
        je      .L3
        cmpl    $0, -4(%ebp)
        je      .L3
        movl    -4(%ebp), %edx
        leal    -8(%ebp), %eax
        addl    %edx, (%eax)
        jmp     .L2

.L3:    movl    -8(%ebp), %eax
        movl    %eax, 4(%esp)
        movl    $.LC1, (%esp)
        call    printf
        leave
        ret
```
Evolution of programming languages, 1980's

- "object-oriented" languages: C++
  - better control of structure of really large programs
    better internal checks, organization, safety
  - a program ("compiler", "translator") converts into assembler or C
  - enormous advantages:
    portable: same program can be translated for different machines
    faster, cheaper hardware helps make this happen

#include <iostream>
main() {
  int num, sum = 0;

  while (cin >> num && num != 0)
    sum += num;
  cout << sum << endl;
}

Evolution of programming languages, 1990's

- "scripting", Web, component-based, ...:
  - Java, Perl, Python, Visual Basic, Javascript, ...
  - write big programs by combining components already written
  - often based on "virtual machine": simulated, like fancier toy computer
  - enormous advantages:
    - portable: same program can be translated for different machines
    - faster, cheaper hardware helps make this happen

```javascript
var sum = 0, num;  // javascript
num = prompt("Enter new value, or 0 to end")
while (num != 0) {
    sum = sum + parseInt(num)
    num = prompt("Enter new value, or 0 to end")
}
alert("Sum = " + sum)
```
Evolution of programming languages, 2000's and beyond

• more of the same
  - more specialized languages for specific application areas
    Flash/Actionscript for animation in web pages
  - ongoing refinements / evolution of existing languages
    C, C++, Fortran, Cobol all have new standards in last few years

• copycat languages
  - Microsoft C# strongly related to Java
  - scripting languages similar to Perl, Python, et al

• better tools for creating programs without as much programming
  - mixing and matching components from multiple languages
Why so many programming languages?

• every language is a tradeoff among competing pressures
  - reaction to perceived failings of others; personal taste

• notation is important
  - "Language shapes the way we think and determines what we can think about."
    - Benjamin Whorf
  - the more natural and close to the problem domain, the easier it is to get the machine to do what you want

• higher-level languages hide differences between machines and between operating systems

• we can define idealized "machines" or capabilities and have a program simulate them -- "virtual machines"
  - programming languages are another example of Turing equivalence