GOALS OF THE COURSE

• Learn fundamental structure of programming languages.
• Understand key issues in language design and implementation.
• Increase awareness of the range of available languages and their uses.
• Learn how to learn a new language.
• Get a small taste of programming language theory.
METHOD OF THE COURSE

• Fairly conventional survey textbook, with broad coverage of languages.
• Homework exercises involve programming problems in real languages.
• Most homework problems will involve modifying implementations of “toy” languages that illustrate key features and issues.
• Exercises will use two modern languages: Java and Standard ML.
• Between them, these languages illustrate many of the important concepts in current language designs.
NON-GOALS

- Teaching how to program.
- Teaching how to write programs in any particular languages(s).
- Surveying/cataloging the features of lots of different languages.
- Comprehensive coverage of programming paradigms (e.g., will skip logic and concurrent programming material).
- Will mostly be concerned with interpreting abstract syntax for the toy languages, and will spend very little time on parsing and code generation. (Not a compiler course!)
What languages do you know?

FORTRAN, COBOL, (Visual) BASIC, ALGOL-60, ALGOL-68, PL/I, C, C++, RPG, Pascal, Modula, Oberon, Lisp, Scheme, ML, Haskell, Ada, Prolog, Goedel, Curry, Snobol, ICON, . . .

Don’t forget things like:

scripting languages: perl, tcl, Python, . . .

SQL, other database query languages.

spreadsheet expression languages

text processing languages, tex, awk, etc.

application-specific languages.
Consider a simple (dumb) algorithm for testing primality.

In Java:

```java
public static boolean isprime (int n) {
    // return true if n has no 
    // divisor in interval [2,n-1]
    for (int d = 2; d < n; d++)
        if (n % d == 0)
            return false;
    return true;
}
```
In Standard ML (using a recursive function):

```ml
fun isprime (n:int) : bool = 
  (* return true if n has no 
   divisor in interval [2,n-1] *)
  let
  fun no_divisor (d:int) : bool = 
    (* return true if n has no 
       divisor in interval [d,n-1] *)
    (d >= n) orelse((n mod d <> 0) andalso 
       (no_divisor (d+1)))
  in
  no_divisor 2
  end
```
In Intel X86 Assembler:

.globl isprime
isprime:
pushl %ebp ; set up procedure entry
movl %esp,%ebp
pushl %esi
pushl %ebx
movl 8(%ebp),%ebx ; fetch arg n from stack
movl $2,%esi ; set divisor d := 2
cmpl %ebx,%esi ; compare n,d
jge true ; jump if d >= n

loop: movl %ebx,%eax ; set n into ....
cltd ; ... dividend register
idivl %esi ; divide by d
testl %edx,%edx ; remainder 0?
jne next ; jump if remainder non-0
xorl %eax,%eax ; set ret value := false(0)
jmp done

next: incl %esi ; increment d
cmpl %ebx,%esi ; compare n,d
jl loop ; jump if d < n

true: movl $1,%eax ; set ret value := true(1)
done: leal -8(%ebp),%esp ; clean up and exit
popl %ebx
popl %esi
leave
ret
HIGH-LEVEL LANGUAGES: GENERAL CHARACTERISTICS

- Complex Expressions (Arithmetic, Logical, ...)
- Structured Control Operators (Loops, Conditionals, Cases)
- Composite Types (Arrays, Records, etc.)
- Type Declarations and Type Checking
- Multiple storage classes (global/local/heap)
- Procedures/Functions, with private scope, maybe first-class
- Maybe high-level control mechanisms (Exceptions, Back-tracking, etc.)
- Maybe abstract data types, modules, objects, etc.
**MACHINE CODE CHARACTERISTICS**

- Low-level machine instructions to implement operations.
- Control flow based on labels and conditional branches.
- Explicit locations (e.g. registers) for values and intermediate results.
- Explicit memory management (e.g., stack management for procedures).
A stack machine is a simple architecture based on a stack of operand values.

- All machine instructions pop their operands from the stack, and push their results back onto the stack.
- This makes instructions very simple, because there’s no need to specify operand locations.
- This architecture is often used in abstract machines, such as the Java Virtual Machine. (Most real machines use register-based architectures instead.)
- We will use stack machine programs as (slightly artificial) example of compiled machine code.
Here's the instruction set for a very simple stack machine:

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Stack Before</th>
<th>Stack After</th>
</tr>
</thead>
<tbody>
<tr>
<td>CONST i</td>
<td>s1 ... sn</td>
<td>i s1 ... sn</td>
</tr>
<tr>
<td>LOAD x</td>
<td>s1 ... sn</td>
<td>Vars[x] s1 ... sn</td>
</tr>
<tr>
<td>STORE x</td>
<td>s1 ... sn</td>
<td>s2 ... sn</td>
</tr>
<tr>
<td>PLUS</td>
<td>s1 s2 s3 ... sn</td>
<td>(s1+s2) s3 ... sn</td>
</tr>
<tr>
<td>MINUS</td>
<td>s1 s2 s3 ... sn</td>
<td>(s2-s1) s3 ... sn</td>
</tr>
</tbody>
</table>

Note that STORE x also sets Vars[x] = s1.
And here's a stack machine program corresponding to the simple statement $c = 3 - a + (b - 7)$:

- `CONST 3`
- `LOAD a`
- `MINUS`
- `LOAD b`
- `CONST 7`
- `MINUS`
- `PLUS`
- `STORE c`

Is this code sequence unique?

This illustrates the expressiveness of high-level expressions compared to machine code.
Programming paradigms

- Imperative (including object-oriented)
- Functional
- Logic
- Concurrent/Parallel

Programming Contexts

Programming “in the Small”

- Expressions
- Structured Control Flow
- Structured Data
- Types

Programming “in the Large”

- Modules and Separate Compilation
- Code Re-use; Polymorphism
- Object-oriented Programming
- Types
• Procedural imperative language.
• Widely used for systems programming, especially for Unix, Linux.
• First language to be implemented on almost any new machine.
• Influential expression-oriented syntax.
• Supports “bit-twiddling” low-level manipulation.
• Generates efficient, predictable machine code.
• Weak type system; run-time crashes common.
• Weak structuring mechanisms.
• C++ greatly extends C, but is almost completely backwards-compatible.
- Object-oriented imperative language.
- Originally hyped for “web programming.”
- Designed as a “better C++.”
- Core language largely derived from C,C++.
- Strong static type-checking.
- Classes serve as primary structuring mechanism.
- Automatic heap allocation and garbage collection.
- Extensive libraries.
- Portable interpretive environment (JVM) based on stack abstract machine.
- C# is another, very similar language.
STANDARD ML

- Mostly functional language.
- Designed for symbolic manipulation; inherits from LISP, Scheme.
- Strong, static type inference.
- Simple, uniform, powerful recursive datatype mechanism.
- Recursion is primary control structure.
- Powerful parameterized module mechanism.
- Automatic heap allocation and garbage collection.
- Top-level “read-eval-print” loop for development.
LANGUAGE DESCRIPTION AND DOCUMENTATION

For programmers, compiler-writers, and students . . .

Syntax (Easy)
- Grammars; BNF and Syntax Charts

Semantics (Hard)
- Informal
- Formal: Operational, Denotational, Axiomatic

Learning about a Language
- Reference Manuals
- User Guides
- Texts and tutorials