What is this course about?

- Programming language concepts
  - Syntax
  - Semantics
  - Names, bindings, scopes
  - Data types
  - Expressions and statements
  - Control structures
  - Subprograms
  - Exception and event handling
  - and more...

What is this course about?

- Programming language paradigms
  - Imperative
  - Procedural
  - Object-oriented
  - Functional
  - Logic

Why study this?

- Different languages are designed for different tasks
- Learn how to choose the right language
- Understand implementation issues and language behaviour

Problem Domains

- System programming
  - Low-level
  - Efficiency (space and time)
  - e.g. C
- Scientific Computations
  - Floating-point support
  - Large integer support
  - Matrix/vector
  - e.g. Fortran, APL, MATLAB, Maple, Mathematica
- Business Applications
  - Report generation
  - Binary Coded Decimals
  - e.g. COBOL

Problem Domains

- Artificial intelligence
  - symbolic computation, logic programming
  - self-modifying code
  - e.g. LISP, Scheme, Prolog
- Web
  - Easy to use
  - Support multimedia and multiple platforms
  - e.g. HTML, PHP, Javascript

Evaluating Programming Languages: Readability

- Simplicity
  - number of features
  - multiple meanings (e.g. operator overloading)
- Orthogonality
  - small set of primitives can be combined in few ways to construct many different control and data structures
- Data Types
- Syntax: control structures, reserved words, etc.

Evaluating Programming Lanaguages: Writability

- Simplicity and orthogonality
- Expressivity: number of constructs, what they can do
- Abstraction support: defining new data and control structures

Evaluating Programming Languages: Reliability

- Type checking
- Exception handling
- Aliasing

Cost

- Compiling and executing
- Training
- Development
- Maintenance
- Portability

Implementation

- To be useful, a language must have an implementation.
- Source code can be:
  - compiled
  - interpreted
  - hybrid

Compilation

- A compiler reads the source code, and translates it into machine code.
- The compiler performs the following steps:
  - preprocessing (e.g. `#include`)
  - lexical analysis: separate text into tokens
  - syntax analysis: parses the source code into structural units (often trees)
  - code generation
- A linker combines machine code from multiple source files as well as libraries, and produce an executable
- Translation is slow, execution is fast

Interpretation

- No prior translation
- Source code is translated by an interpreter as it runs
- Usually easier and quicker to modify programs and see the results immediately
- It may even be possible to change the program while it is running
- Execution is slower, generally require more memory space at runtime

Hybrid Implementation

- Source code is first compiled into an intermediate code
- An interpreter reads the byte code and executes on the machine
- Pros: byte code can be platform independent. Platform dependent interpreters are used
- Still slower than compiled code but faster than pure interpretation
- Just-in-time (JIT) compilers: compile byte code before running. Incur overhead to reduce run time.