CMSC 330:
Organization of Programming Languages

Course Logistics

Spring 2026

Course Goals

- Describe and compare programming language features
- Learn some fundamental concepts of Programming Languages
- Choose the right language for the job
- Write better code
  - Code that is shorter, more efficient, with fewer bugs

- In short:
  - Become a better programmer with a better understanding of your tools.

Course Activities

- Learn different **types of languages**
- Learn different **language features**
  - Programming patterns repeat between languages
- Study how languages are **specified**
  - **Syntax**, **Semantics** — mathematical formalisms
- Study how languages are **implemented**
  - Parsing via **regular expressions** (automata theory) and **context free grammars**
  - Mechanisms such as **closures**, **tail recursion**, **type checking**, **lazy evaluation**, **garbage collection**, …

Syllabus

- Functional programming (OCaml)
- Regular expressions & finite automata
- Context-free grammars & parsing
- Lambda Calculus and Operational Semantics
- Safe, “zero-cost abstraction” programming (Rust)
- Scoping, type systems, parameter passing, comparing language styles; other topics

Calendar / Course Overview

- Tests
  - 4 quizzes, 2 midterm exams, 1 final exam
  - Do not schedule your interviews on exam dates
- Lecture quizzes
  - Weekly ELMS quizzes
- Projects
  - Project 0 – Out already!
  - Project 1 - OCaml Basics
  - Project 2,3,4,5 OCaml
  - Project 6,7 Rust projects

- Syllabus: https://bakalian.cs.umd.edu/cmsc330/syllabus

Grading

- Assessment in this course consists of proctored examinations and unproctored projects.
  - Proctored:
    - Midterms 1 and 2: 10% each
    - Final: 22%
    - 4 quizzes: 20%
  - Unproctored:
    - 5 Projects: 34%
    - Gradescope quizzes: 4%
- Benchmark Grade Requirements for Proctored Components:
  - Obtain an average of 60% across all exams and quizzes

Discussion Sections

- Discussions will be in-person
- Discussion sections will deepen understanding of concepts introduced in lecture
- Oftentimes discussion section will consist of programming exercises
- There will also be be quizzes, and some lecture material in discussion section

Project Grading

- Projects will be graded using the Gradescope
  - Software versions on these machines are canonical
- Submit often. Activate the best submission
- Develop programs on your own machine
  - Your responsibility to ensure programs run correctly on gradescope
- See web page for OCaml, Rust versions we use, if you want to install at home

Rules and Reminders

- Lectures will be recorded.
- Use lecture notes as your text
  - You will be responsible for everything in the notes, even if it is not directly covered in class!
- Keep ahead of your work
  - Get help as soon as you need it
  - Office hours, Piazza (email as a last resort)
- Avoid distractions, to yourself and your classmates
  - Keep cell phones quiet

Academic Integrity

- All written work (including projects) done on your own
  - Do not copy code from other students or from the web
  - Do not post your code on the web
- Cheaters are caught by auto-comparing code
- The use of of AI generated code like chatGPT or Github Copilot is a violation of the Academic Integrity Policy
- Work together on *high-level* project questions
  - Discuss approach, pointers to resources: OK
- Work together on practice exam questions

CMSC 330:
Organization of Programming Languages

Overview

Plethora of programming languages

- Java, C/C++, C#
- LISP, Scheme, Racket, Haskell, OCaml
- Python, Ruby, JavaScript
- More

All Languages are (sort of) Equivalent

- A language is Turing complete if it can compute any function computable by a Turing Machine

- Essentially all general-purpose programming languages are Turing complete
  - I.e., any program can be written in any programming language

Studying Programming Languages will make you a better programmer

- Ideas or features from one language translate to, or are later incorporated by, another
  - Many "design patterns" in Java are functional programming techniques
- Learn to distinguish surface differences from deeper principles
  - Features of a language make it easier or harder to program for a specific application
- Using the right programming language or style for a problem may make programming:
  - Easier, faster, less error-prone

Studying Programming Languages
Become better at learning new languages

- A language not only allows you to express an idea, it also shapes how you think when conceiving it

- You may need to learn a new (or old) language
  - Paradigms and fads change quickly in CS
  - Also, may need to support or extend legacy systems

Changing Language Goals

- 1950s-60s – Compile programs to execute efficiently
  - Language features based on hardware concepts, Integers, reals, `goto` statements
  - Programmers cheap; machines expensive
    - Computation was the primary constrained resource
  - Programs had to be efficient because machines weren't
    - Note: this still happens today, just not as pervasively

Changing Language Goals: Now

- Program complexity has increased.
- Developer productivity matters more than raw speed.
  - Hardware is much faster and cheaper, trade performance for clearer, shorter, and safer code.
- Correctness and safety are more important.
  - financial systems, medical devices
- Concurrency and distribution are now central concerns.
- Security is a first-class goal.

Programming in the Age of AI

- From writing code to describing intent.
  - state *what* a program should do, not exactly *how* to do it.
- Languages as interfaces to AI systems.
  - Like we interface with operating systems now
- Greater emphasis on correctness and verifiability.
  - strong static typing
  - formal specifications
  - test generation
- More automation, less boilerplate.
  - from “write everything explicitly” → “generate from patterns and intent”
- Adaptation and evolution at runtime.
  - Future systems may include learning components that change behavior over time.

Language Attributes to Consider

- Syntax
  - What a program looks like
- Semantics
  - What a program means (mathematically), i.e., what it computes
- Paradigm and Pragmatics
  - How programs tend to be expressed in the language
- Implementation
  - How a program executes (on a real machine)

Syntax

- The keywords, formatting expectations, and structure of the language
  - Differences between languages usually superficial
    - C / Java `if (x == 1) { … } else { … }`
    - Ruby `if x == 1 … else … end`
    - OCaml `if (x = 1) then … else …`
  - Differences initially jarring; overcome with experience
- Concepts such as **regular expressions**, **context-free grammars**, and **parsing** handle language syntax

Semantics

- What does a program *mean*? What does it *compute*?
  - Same syntax may have different semantics in different languages!

	Physical Equality	Structural Equality
Java	a == b	a.equals(b)
C	a == b	*a == *b
Ruby	a.equal?(b)	a == b
OCaml	a == b	a = b

- Can specify semantics informally (in prose) or formally (in mathematics)

Formal (Mathematical) Semantics

- What do my programs mean?

```
let rec fact n =
  if n = 0 then 1
  else n * (fact n-1)
```

```
let fact n =
  let rec aux i j =
    if i = 0 then j
    else aux (i-1) (j*i) in
  aux n 1
```

- Both OCaml functions implement "the factorial function." How do I know this? Can I prove it?
  - Key ingredient: a mathematical way of specifying what programs do, i.e., their semantics
  - Doing so depends on the semantics of the language

Paradigm

- There are many ways to compute something
  - Some differences are superficial
    - For loop vs. while loop
  - Some are more fundamental
    - Recursion vs. looping
    - Mutation vs. functional update
    - Manual vs. automatic memory management
- Language's paradigm favors some computing methods over others.
- This course uses OCaml and Rust as vehicles for exploring these concepts.

Defining Paradigm: Elements of PLs

- Important features
  - Regular expression handling
  - Objects
    - Inheritance
  - Closures/code blocks
  - Immutability
  - Tail calls
  - Pattern matching
    - Unification
  - Abstract types
  - Garbage collection
- Declarations
  - Explicit
  - Implicit
- Type system
  - Static
    - Polymorphism
    - Inference
  - Dynamic
  - Type safety

Summary

- Programming languages differ in their
  - syntax,
  - semantics,
  - style and paradigm, pragmatics,
  - implementation strategies.
- Each language is designed with particular goals in mind, and these goals evolve as the computing environment changes.
- Concepts developed in one language frequently influence the design of others.
- Therefore, our focus is on learning transferable concepts and skills rather than any single programming language.