CMSC 330:
Organization of Programming Languages

Course Logistics

Fall 2026

Important Links

- **Class Website**:
  - https://www.cs.umd.edu/class/fall2026/cmsc330-01XX
- **Gradescope**: www.gradescope.com
- **ELMS**: https://elms.umd.edu/
- **Gitlab**: gitlab.cs.umd.edu
- **Prairietest**: https://us.prairietest.com/
- **PrairieLearn**: https://www.prairielearn.com/
- **Clicker**: https://buzzer.cs.umd.edu

Course Goals

- Describe and compare programming language features
- Learn some fundamental concepts of Programming Languages
- 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

Coursework

- Tests
  - 6 quizzes, 2 midterm exams, 1 final exam
  - Do not schedule your interviews on exam dates
- Clicker quizzes
  - Every lecture
- Projects
  - 6 or 7 projects
  - Project 1 - OCaml Basics
  - Project 2,3,4,5 OCaml
  - Project 6,7 Rust projects

Exam and Quiz Dates

Assessment	Date
Quiz 1	September 11
Quiz 2	September 25
Midterm 1	**October 8**
Quiz 3	October 16
Quiz 4	October 30
Quiz 5	November 13
Midterm 2	**November 19**
Quiz 6	December 4
Final Exam	**December 15, 6:30pm – 8:30pm**

Grading

Component	Percentage	Date
Assignments	30%	Weekly
Clicker Quizzes	5%	Every lecture
Quizzes(6)	18%	Biweekly
Midterm 1	12%	October 8
Midterm 2	12%	November 19
Final Exam	22%	December 15
Meet the Professor	1%	Before December 4

Clicker quizzes are graded on correctness, and we drop 20% of the clicker questions given over the semester

Exam and Quizzes

- Two midterms and six quizzes, all **in person, proctored**, taken on your own computer through **PrairieTest.**

- Exams are **75 minutes;** quizzes are **50 minutes.**

- You get **two attempts** at every exam and quiz
  - Reserve a second sitting on PrairieTest, then retake it in the exam room
  - Questions are randomized: a similar but different set
  - The grade from your **most recent attempt** counts — not the highest!

- Exam room hours:
  - Monday, Wednesday, Friday 12:00–1:50pm, CSI 2107
  - Tuesday, Thursday 5:00–6:00pm, CSI 2117

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
- Work together on *high-level* project questions
  - Discuss approach, pointers to resources: OK
- Work together on practice exam questions

AI Usage Policy

- You are **encouraged** to use AI tools as learning assistants, tutors, and TAs
  - Ask questions, clarify difficult concepts, review your work, find errors, get extra explanations and examples
- You must implement all assignments and projects yourself
  - AI may not generate, substantially write, or implement a solution on your behalf
- Project material is assessed on proctored exams and quizzes
  - You may be asked to explain, modify, reproduce, or apply concepts from your projects
  - Relying on AI without understanding the material means losing credit there
- Ultimately, you are responsible for knowing and understanding the code in your assignments

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

`mov` is Turing-complete

Stephen Dolan
Computer Laboratory, University of Cambridge
stephen.dolan@cl.cam.ac.uk

Abstract

It is well-known that the `x86` instruction set is baroque, overcomplicated, and redundantly redundant. We show just how much fluff it has by demonstrating that it remains Turing-complete when reduced to just one instruction.

The instruction we choose is `mov`, which can do both loads and stores. We use no unusual addressing modes, self-modifying code, or runtime code generation. Using just this instruction (and a single unconditional branch at the end of the program to make nontermination possible), we demonstrate how an arbitrary Turing machine can be simulated.

registers for now, but later we show how their number can be reduced without losing expressiveness.

We have the following instructions (if you like RISC) or addressing modes (if you like CISC). We use Intel `x86` syntax, where the `mov` instructions have the destination first and the source second, and square brackets indicate memory operands.

Instruction	`x86` **syntax**
Load Immediate	`mov` $R_{\texttt{dest}}$`, c`
Load Indexed	`mov` $R_{\texttt{dest}}$`, [`$R_{\texttt{src}}$ `+` $R_{\texttt{offset}}$`]`
Store Indexed	`mov [`$R_{\texttt{dest}}$ `+` $R_{\texttt{offset}}$`],` $R_{\texttt{src}}$

On `x86`, these are all addressing modes of the same `mov` in-

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

Programming in the Age of AI

Main stages of the Software Development Life Cycle (SDLC)

- Planning and feasibility
- Requirements analysis
- Design
- Development (coding) ← Becoming less important
- Testing
- Deployment
- Maintenance

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.