CMSC 330:
Organization of Programming Languages

Course Logistics

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.

About Me

- I am a Uyghur. Google Uyghur to learn more.
- I joined UMD CS in 2015. CMSC330 is my favorite class. I taught CMSC330 every semester from 2015 to 2021.

What about you?

Ask people on both sides of you

- Name
- Hometown
- Classes they are taking?
- Favorite programming language

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

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
- 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

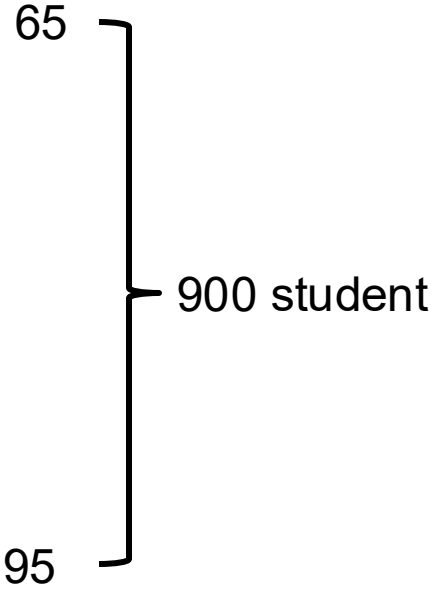
- Lectures will be recorded.
- Use lecture notes as your text
  - You will be responsible for everything in the notes, even if it is not directly covered in class!
- Keep ahead of your work
  - Get help as soon as you need it
  - Office hours, Piazza (email as a last resort)
- Avoid distractions, to 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
  - Do not copy code 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
  - Do not look at/describe another student's code
  - If unsure, ask an instructor!
- Work together on practice exam questions

About the Final Letter Grades

- Each point corresponds to about 30 students. Typically, there are more than 10 students between your grade and the cutoff. Please do not email me requesting a grade bump.



CMSC 330:
Organization of Programming Languages

Overview

Plethora of programming languages

- LISP:
```
(defun double (x) (* x 2))
```
- Prolog:
```
size([],0).
size([H|T],N) :- size(T,N1), N is N1+1.
```
- OCaml:
```
List.iter (fun x -> print_string x)
                    ["hello, "; s; "!\n"]
```
- Smalltalk:
```
( #( 1 2 3 4 5 ) select:[:i | i even ] )
```

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

- Therefore this course is useless?!
  - Learn one programming language, always use it

Studying Programming Languages

- Will make you a better programmer
  - Programming is a human activity
    - Features of a language make it easier or harder to program for a specific application
  - 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
  - Using the right programming language or style for a problem may make programming:
    - Easier, faster, less error-prone

Studying Programming Languages

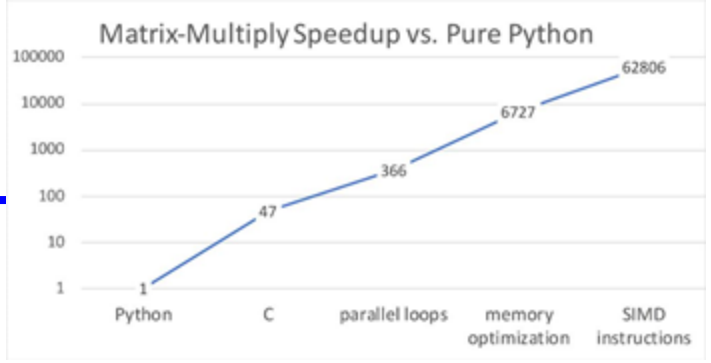
- Become better at learning new languages
  - A language not only allows you to express an idea, it also shapes how you think when conceiving it
  - You may need to learn a new (or old) language
    - Paradigms and fads change quickly in CS
    - Also, may need to support or extend legacy systems

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



- Today
  - Language features based on design concepts
    - Encapsulation, records, inheritance, functionality, assertions
  - Machines cheap; programmers expensive
    - Scripting languages are slow(er), but run on fast machines
    - They've become very popular because they ease the programming process
  - The constrained resource changes frequently
    - Communication, effort, power, privacy, …
    - Future systems and developers will have to be nimble

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 class:
  - Imperative
  - Functional
  - Resource-controlled (zero-cost)
  - Scripting/dynamic

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

Imperative Languages

- Also called procedural or von Neumann
- Building blocks are procedures and statements
  - Programs that write to memory are the norm

```
int x = 0;
while (x < y) x = x + 1;
```

  - FORTRAN (1954)
  - Pascal (1970)
  - C (1971)

Functional (Applicative) Languages

- Favors immutability
  - Variables are never re-defined
  - New variables a function of old ones (exploits recursion)
- Functions are higher-order
  - Passed as arguments, returned as results

  - LISP (1958)
  - ML (1973)
  - Scheme (1975)
  - Haskell (1987)
  - OCaml (1987)

OCaml

- A (mostly-)functional language
  - Has objects, but won't discuss (much)
  - Developed in 1987 at INRIA in France
  - Dialect of ML (1973)
- Natural support for pattern matching
  - Generalizes `switch`/`if-then-else` – very elegant
- Has full featured module system
  - Much richer than interfaces in Java or headers in C
- Includes type inference
  - Ensures compile-time type safety, no annotations

Zero-cost Abstractions in Rust

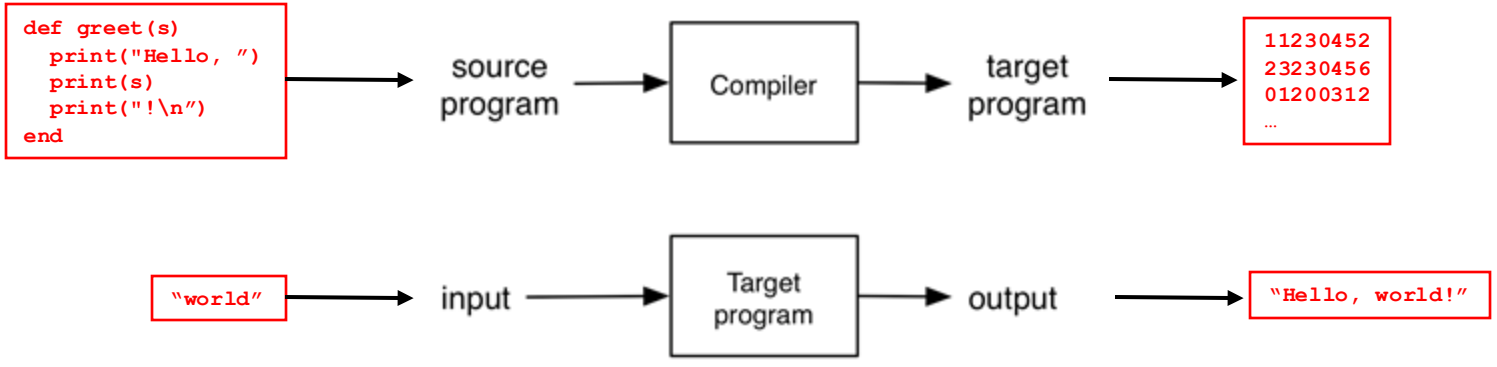
- A key motivator for writing code in C and C++ is the low (or zero) cost of the abstractions use
  - Data is represented minimally; no metadata required
  - Stack-allocated memory can be freed quickly
  - Malloc/free maximizes control – no GC or mechanisms to support it are needed
- But no-cost abstractions in C/C++ are insecure
- Rust language has safe, zero-cost abstractions
  - Type system enforces use of ownership and lifetimes
  - Used to build real applications – web browsers, etc.

Implementation

- How do we implement a programming language?
  - Put another way: How do we get program P in some language L to run?

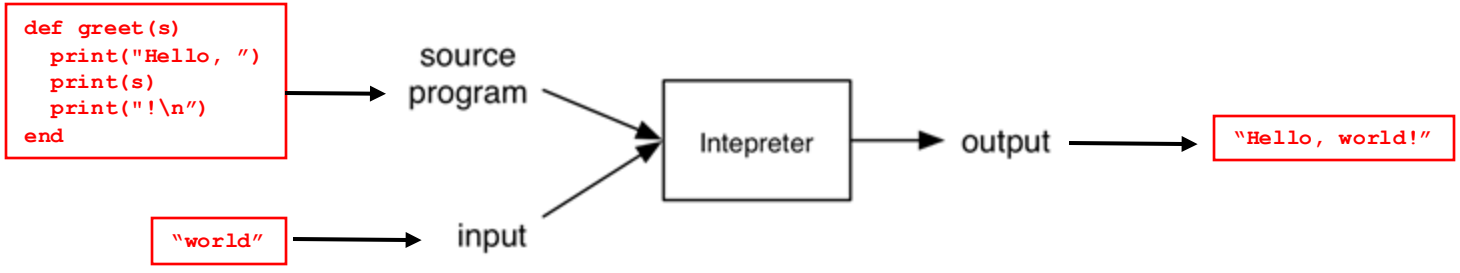
- Two broad ways
  - Compilation
  - Interpretation

Compilation



- Source program translated ("compiled") to another language
  - Traditionally: directly executable machine code
    - gcc, clang
  - Bytecode, Portable Code
    - Javac

Interpretation



- Interpreter executes each instruction in source program one step at a time
  - No separate executable

Summary

- Programming languages vary in their
  - Syntax
  - Semantics
  - Style/paradigm and pragmatics
  - Implementation
- They are designed for different purposes
  - And goals change as the computing landscape changes, e.g., as programmer time becomes more valuable than machine time
- Ideas from one language appear in others

OCaml Compiler

- OCaml programs can be compiled using ocamlc
  - Produces .cmo ("compiled object") and .cmi ("compiled interface") files
    - We'll talk about interface files later
  - By default, also links to produce executable a.out
    - Use -o to set output file name
    - Use -c to compile only to .cmo/.cmi and not to link
- Can also compile with ocamlopt
  - Produces .cmx files, which contain native code
  - Faster, but not platform-independent (or as easily debugged)

OCaml Compiler

- Compiling and running the following small program:

hello.ml:

```
(* A small OCaml program *)
print_string "Hello world!\n";;
```

```
% ocamlc hello.ml
% ./a.out
Hello world!
```

OCaml interpreter

```
% ocaml hello.ml
Hello world!
```

OCaml Compiler: Multiple Files

main.ml:

```
let main () =
  print_int (Util.add 10 20);
  print_string "\n"

let () = main ()
```

util.ml:

```
let add x y = x+y
```

- Compile both together (produces `a.out`)

```
ocamlc util.ml main.ml
```

- Or compile separately

```
ocamlc -c util.ml
ocamlc util.cmo main.ml
```

- To execute

```
./a.out
```

OCaml Top-level

- The *top-level* is a read-eval-print loop (REPL) for OCaml
- Start the top-level via the **`ocaml`** command

```
#ocaml
% OCaml version 4.14.1
print_string "Hello world!\n";;
Hello world!
```

- **`utop`** is an alternative top-level; improves on **`ocaml`**
- To exit the top-level, type ^D (Control D) or call the exit 0

```
exit 0;;
```

OCaml Top-level

Expressions can be typed and evaluated at the top-level

```
# 3 + 4;;
- : int = 7

# let x = 37;;
val x : int = 37

# x;;
- : int = 37

# let y = 5;;
val y : int = 5

# let z = 5 + x;;
val z : int = 42

# print_int z;;
42- : unit = ()

# print_string "Colorless green ideas sleep furiously";;
Colorless green ideas sleep furiously- : unit = ()

# print_int  "Colorless green ideas sleep furiously";;
This expression has type string but is here used with type int
```

Loading Code Files into the Top-level

File **hello.ml** :

```
print_string "Hello world!\n";;
```

- Load a file into top-level

  #use *"filename.ml"*

- Example: ← #use processes a file a line at a time

```
# #use "hello.ml";;
Hello world!
- : unit = ()
#
```

OPAM: OCaml Package Manager

- **`opam`** is the package manager for OCaml
  - Manages libraries and different compiler installations

- You should install the following packages with **`opam`**
  - **`ounit`**, a testing framework similar to minitest
  - **`utop`**, a top-level interface
  - **`dune`**, a build system for larger projects

Project Builds with **dune**

- Use **dune** to compile projects---automatically finds dependencies, invokes compiler and linker
- Define a **dune** file, similar to a **Makefile**:

dune:

```
(executable
 (name main))
```

Indicates that an executable (rather than a library) is to be built

Name of main file (entry point)

```
% dune build main.exe
% _build/default/main.exe
30
%
```

Check out https://medium.com/@bobbypriambodo/starting-an-ocaml-app-project-using-dune-d4f74e291de8

Dune commands

- If defined, run a project's test suite:

```
dune runtest
```

- Load the modules defined in src/ into the **`utop`** top-level interface:

```
dune utop src
```

- **`utop`** is a replacement for **`ocaml`** that includes dependent files, so they don't have be be **`#load`**ed

A Note on ;;

- ;; ends an expression in the top-level of OCaml
  - Use it to say: "Give me the value of this expression"
  - Not used in the body of a function
  - Not needed after each function definition
    - Though for now it won't hurt if used there

- There is also a single semi-colon ; in OCaml
  - But we won't need it for now
  - It's only useful when programming imperatively, i.e., with side effects
    - Which we won't do for a while