CMSC330: Introduction

Chris Kauffman

Last Updated:
Tue Aug 29 09:15:05 AM EDT 2023

Logistics

Slides: https://kauffman77.github.io/330/

Introductions

CMSC330 Sec 3xx Prof TBA = **Prof Kauffman**

- kauffman77@gmail.com for now
- Office Hours Tue/Wed 1-2pm in IRB2226

All CMSC330 Sections Coordinated

- Our section will remain in sync with 1xx/2xx sections
- Same Syllabus, HW, Quiz/Exam Schedules

Reading: None

Goals

- Get The Lay of the PL Landscape
- Survey the course content and structures
- Sample Course Structure / Mechanics


C, C++, C#
D, Rust, Julia
Imperative
Ruby, R
Javascript, Lua
Java
Object
Python
Static
Compile
CMSC 330
Eval
Dynamic
OCaml
Closure
Racket
Haskell, SML, F#
Prolog
Functional
Scheme, Clojure
Lisp

Organization of Diagram

Typing Discipline (Horizontal Axis)

Static Typing : Variables are bound to the same type of data over their lifetime

Dynamic Typing : Variables may be bound to data of differing types over their lifetime

Mutability of Data (Vertical Axis)

Imperative Paradigm : Emphasize variables with values that change variables change over time, reflect the nature of the Turing Machine

Functional Paradigm : Favor immutable data, bindings to values are fixed but new versions may be created, reflect the nature of the Lambda Calculus

Cross-Cutting Concerns for PLs

Built-in Data Does the PL feature built-in data types like arrays, linked lists, sets, and maps (e.g. hash tables)? Is there a handy syntax to use these?

Code Organization How does the PL allow code to be divided into **modules** and expose some/all of their contents? e.g. Names and their Namespaces

Translation Model Does the PL support ahead of time **compilation** to some lower form or piece-by-piece **interpretation**? Or does it does it support something in between?

Dynamic Execution Does the language allow execution of new, unknown code at runtime? The creation of new code within the current program?

Abstraction Facilities How does the PL allow for patterns and ideas to be encoded? Can its syntax be extended?

Object Orientedness Does the PL support OO programming, in whole or part?

Exercise: Object Orientendness of Languages

- CMSC330 students should have at least 3 semesters of college coding courses such as CMSC131, 132, and 216
- Undoubtedly you'll have been exposed to **Object Oriented Programming**, a style of programming popularized during 1990s and still prevalent today

Consider:

1. Name an OO language you have used and whether like it
2. What makes that or any Language object-oriented?
3. What makes a Program object-oriented, irrespective of PL?
4. What is Object Oriented Programming exactly?

Chat up your neighbors, consider some ideas, we'll regroup and share together momentarily.
A few folks that share answers will be granted a small amount of bonus credit at the end of the semester

Answers: Object-Orientendness of Languages 1/3

1. Name an OO language you have used and whether like it

   Java will be the most common for UMD students. Python, C++, and Javascript / ECMAScript are also common for those starting out.

2. What makes that or any Language object-oriented?

   Typically has features that ease the creation of object oriented programs. Examples include built-in syntax for classes, inheritance, and ***dynamic dispatch****.*

   Dynamic dispatch *is automatically selecting and executing one of several versions of a function based on they type of the data passed as a parameter.*

   HOWEVER methods and dynamic dispatch are not unique to OO programs. Languages like Java, C++, and Python support ***single dynamic dispatch*** *with syntax like* `animal.speak(msg)`*. We may look at other languages that support* ***multiple dynamic dispatch*** *such as Racket, Clojure, and Julia, none of which are object-oriented.*

Answers: Object-Orientendness of Languages 2/3

3. What makes a Program object-oriented, irrespective of PL?

   Typically models its domain by dividing it into classes and defining methods that act on those classes. ***However*** *ALL sensible programs. . .*

   - *Define data types for their domain*
   - *Define functions that act on those data types*
   - *Hide aspects of implementation through abstractions to allow future flexibility*
   - *Establish general-case functionality that can be re-used and adapted to expand and handle new situations*

   So, these are NOT OO in their own right.

   Use of ***class hierarchies with inheritance*** *are the most likely features to make a Program OO though opinions vary on the wisdom of using classes inheritance.*[1] *However, hierarchical data types are not unique to OO programs.*

[1] Noted OO writer Allen Holub has an article on the difficulties of class hierarchies. He mentions that even James Gosling, the primary author of Java, indicated he would leave out inheritance if allowed to re-design Java.

Answers: Object-Orientendness of Languages 3/3

4. What is Object-Oriented Programming exactly?

   If you ever figure this out, let me know as I'm at year 20 of investigating and don't yet have a more satisfactory answer than "buzz word".

Several of the languages we will discuss support Object-Oriented styles of programming. We will touch on these a little going forward but also describe alternatives to the Kingdom of Nouns.

The 4 Exemplars: Imperative Players

Java

- Imperative (mostly)
- Statically typed (mostly)
- Object-Oriented (questionably)
- Ubiquitous in industry
- Covered in prereq courses, used as a comparison point for others

If I have to write `public static void main()...` one more time I'm gonna puke

Python

- Imperative (mostly)
- Dynamically typed (and regretting it)
- Object-Oriented (haphazardly)
- Widely used in everything from data science to robotics to back end development
- Covered early in course, a great resume padder

Prepare for `self` -loathing. (see)

The 4 Examplars: Functional Players

OCaml

- Functional (but not rigidly)
- Statically typed (rigidly)
- Can be OO, but not necessarily so (Objective-CAML)
- Not widely used but its ML family has inspired many newer languages
- Covered midway through course

Because it's French, none of the controls are in the usual places. (src)

Racket

- Functional (but not rigidly)
- Dynamically typed (default)
- Can be OO, but not necessarily so `(require racket/class)`
- Of the LISP family, understood by few, imitated by all
- Covered near end of course

These are your father's parens, elegant weapons from a more civilized time. (src)

Exercise: Exemplars Sum 1 to 10, Imperative Style

- Each produces the output `x is 50`
- Examine the code with some neighbors; identify similarities and differences
- What technique is used to produce the output in each case?

```java
// SumLoop.java: Java
// statically typed, imperative
public class SumLoop {
  public static void main(String args[]){
    int x = 0;
    for(int i=1; i<=10; i++){
      x = x+i;
    }
    System.out.printf("x is %d\n",x);
  }
}
```

```python
# sumloop.py: Python
# dynamically typed, imperative
x = 0
for i in range(1,11):
  x = x+i
print("x is {}".format(x))
```

```ocaml
(* sumloop.ml: OCaml
   statically typed, functional *)
let _ =
  let x = ref 0 in
  for i=1 to 10 do
    x := !x + i;
  done;
  Printf.printf "x is %d\n" !x;
```

```racket
;; sumloop.rkt: Racket
;; dynamically typed, functional
#lang racket/base
(define x 0)
(for ([i 11])
  (set! x (+ x i)))
(printf "x is ~v\n" x)
```

Answers: Exemplars Sum 1 to 10, Imperative Style

A few things you might observe aside from syntax differences

- Java and Python use the familiar `x = ...` to assign new values to `x` while OCaml and Racket are a bit different
- Despite OCaml being labeled **statically typed** there are no types like `int` listed in the program; *interesting...*
- Only Java uses curly braces for constructs; the others have alternatives; Java also requires a surrounding class to be used
- Each language uses **formatted output** to create a string with variable values substituted in; hopefully you've seen `printf()` before and are a bit familiar with this idea as it will recur

Codes are in **today's codepack**: `01-introduction-code.zip` linked next to lecture slides. It's probably a good idea to...

- Download the codepack and poke around
- Set up your environment so that you can execute each code (this is the essence of Project 0)

Rust: The Kid Sibling PL

- Covered at 2/3 point in Course
- General purpose programming environment combining aspects of OCaml/ML and C
- Programs adhere to a complex set of rules on data **ownership** that allow Compiler to recoup memory automatically, prevent certain errors, and execute code with reasonable efficiency without garbage collection
- Originated at Mozilla, first release in 2015, experimental support within Linux Kernel for Drivers


```
1 // sumloop.rs
2 fn main() {
3   let mut x = 0;
4   for i in 1..11 {
5     x = x+i;
6   }
7   print!("x is {}\n",x);
8 }
```

Rust is named after a fungus. It only gets worse from there.

Rustlings want Rust to replace C for systems programming. C programmers are presently pushing aside the dead husks of past PLs that have tried and failed to unseat C so they can see what Rust has on offer.

Underlying Theory and Technologies of PL

> *All of these PLs suck. I will make the One PL to rule them all, and in the darkness bind them. Though I don't know the way...*

While examining attributes of programming languages, we'll also explore underlying theory and technique tied to PLs including...

PL Theory/Tech: Finite State Machines

Models how formal languages can be recognized, used to build implementations of regular expressions. Variants include

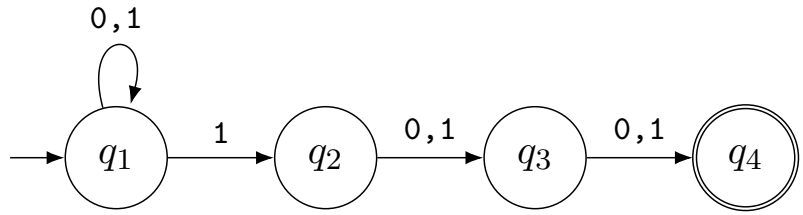
- Non-deterministic Finite Automata: smaller but require backtracking
- Deterministic Finite Automata: larger but no backtracking req'd

Study the theory of these and how they are used to create code for language processing

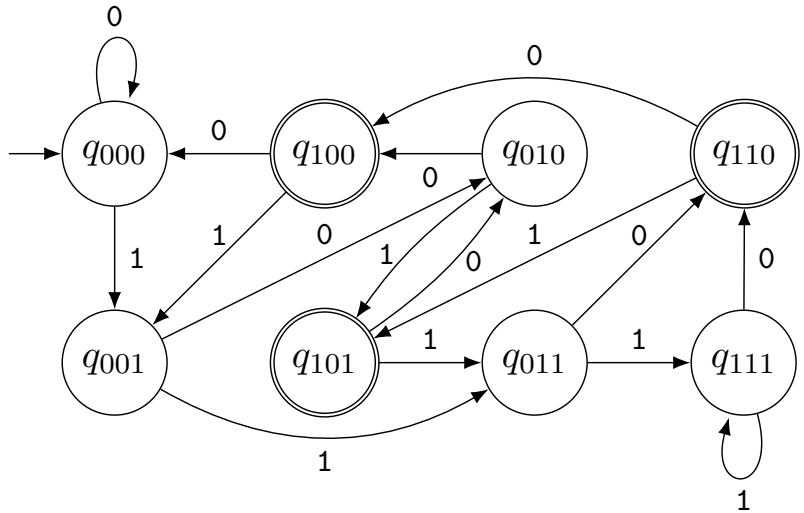
Regular Language

`A = (0|1)*1(0|1)(0|1)`

NFA Recognizing A

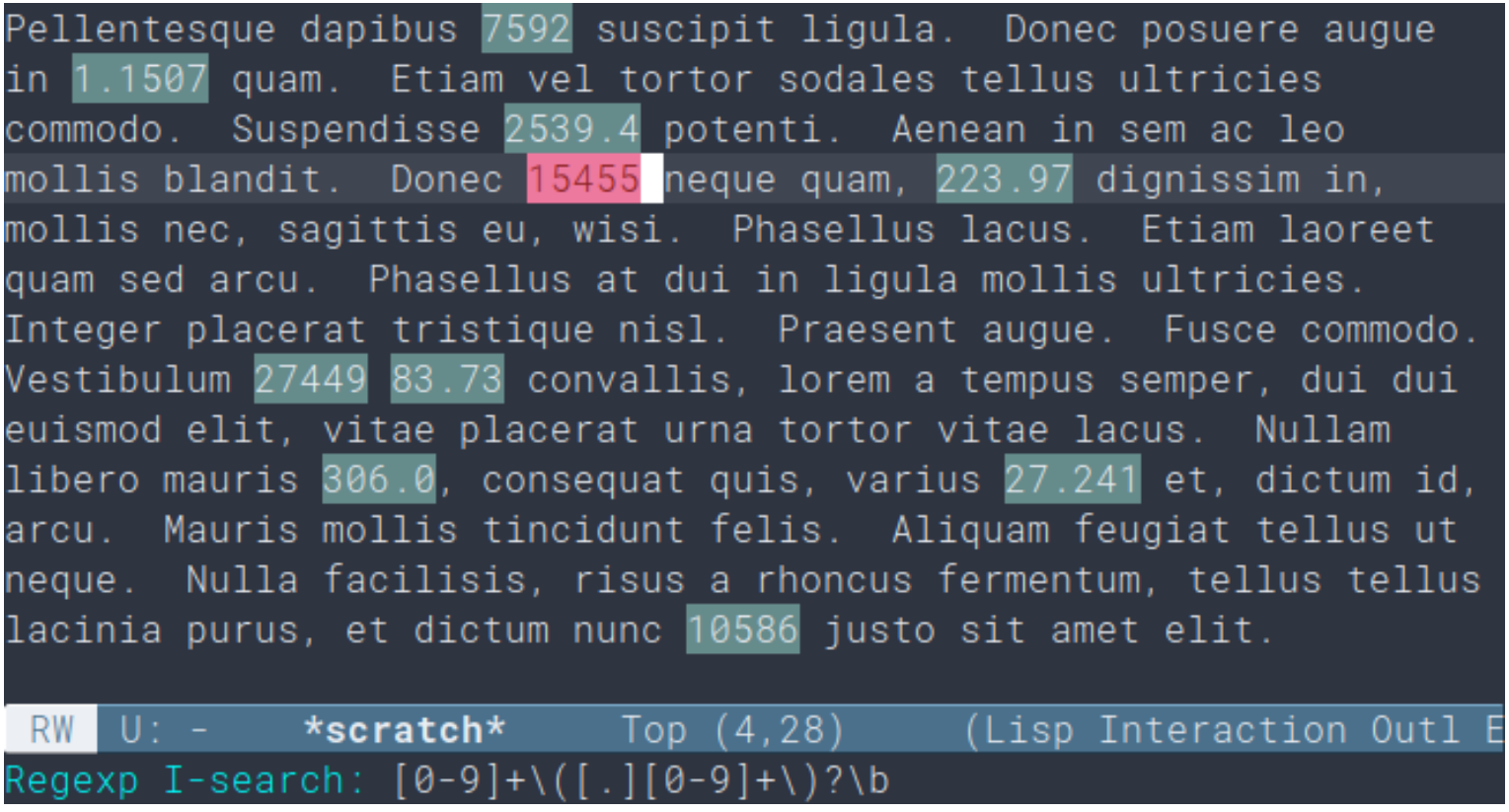


DFA Recognizing A



PL Theory/Tech: Regular Expressions

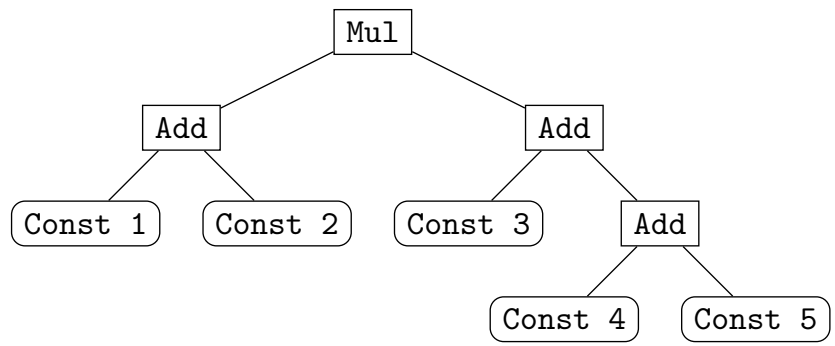
The tool of choice for complex string matching and replacement. An underlying theory and a mini-language in every PL worth its salt. Indispensable in editors, on the command line, and in language processing.



PL Theory/Tech: Lexing and Parsing

- Processing the raw input text of a program into a data structure that can be either directly interpreted or compiled to some other form like Assembly Language
- Can hand-roll code for this but tools in the vein of `Lex / YACC` automate the process and are available for a variety of PLs allowing one to build mini-languages

```
>> ocaml
OCaml version 5.0.0
# #load "langproc.cmo";;
# open Langproc;;
# let l = lex_string "(1+2) * (3+4+5)";;
l : Langproc.token list =
[OParen; Int 1; Plus; Int 2;
 CParen; Times; OParen;
 Int 3; Plus; Int 4; Plus;
 Int 5; CParen]
# parse_tokens l;;
- : Langproc.expr =
Mul (Add (Const 1, Const 2),
     Add (Const 3, Add (Const 4,
                        Const 5)))
```



PL Theory/Tech: The Lambda Calculus

- A theory of computation equivalent in power to **Turing Machines**
- The basis for functional PLs like Racket, Scheme, Lisp, OCaml, ML, etc.
  - There's an Untyped Lambda Calculus and a bunch of Typed Lambda Calculi
  - You can probably guess which one inspired dynamically and statically typed functional PLs
- It's source of all those pesky `lambda`'s that show up in PLs.

The Church Encoding for Pairs in Lambda Calculus

$$PAIR := \lambda x.\lambda y.\lambda f.fxy$$

$$FIRST := \lambda p.p\text{TRUE}$$

$$SECOND := \lambda p.p\text{FALSE}$$

$$NIL := \lambda x.\text{TRUE}$$

$$NULL := \lambda p.p(\lambda x.\lambda y.\text{FALSE})$$

Why Study Programming Languages?

If the only tool you have is a hammer, it is tempting to treat everything as if it were a nail.
– *Abraham Maslow*

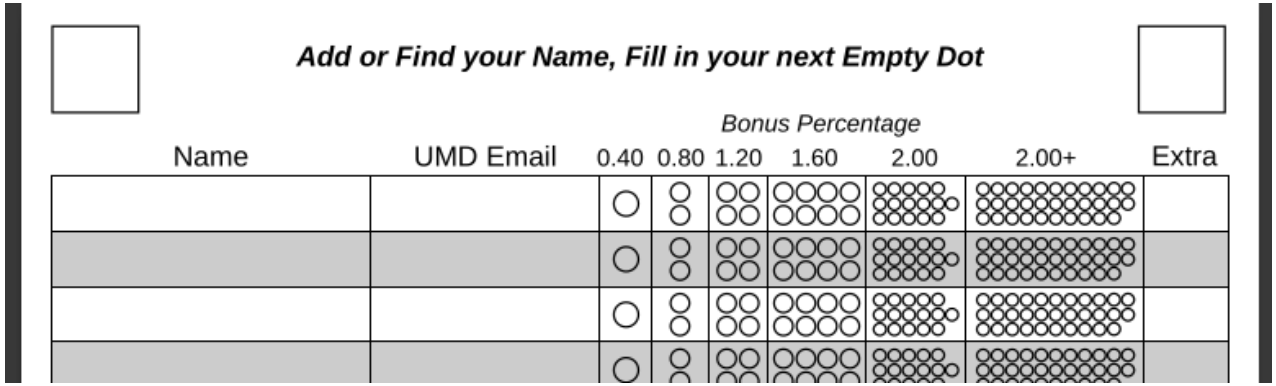
- Programming Languages are Tools made by people for people to do computations
- Tools have trade-offs, are better at some jobs and worse at others
- **Hackers** hammer everything they see and justify it by loudly proclaiming their hammer as the superior tool
- **Wizards** skip the debate, quietly select the appropriate tool among their many options, get the job done faster, and go home early
- Strive to be a Wizard

Course Structure

Perc	Element	Notes
40%	8 Projects	Individual work; credit varies per project complexity
24%	2 Midterm Exams	In lecture on posted dates
22%	Final Exam	Common Exam w/ other sections
10%	4 Discussion Quizzes	In discussion on posted dates
4%	Lecture quizzes	Planned weekly, online
2%	Bonus Credit	For participation in Lec/Disc

- 4xx Lectures will be recorded and made available via ELMS/Canvas links (as soon as Kauffman can get on Canvas)
- Lecture slides/codepacks will be posted and linked from the course website

In-Class Participation for Bonus Credit



- If you engaged with lecture or discussion by offering an answer to an in-class system or asked an interesting question, come up at the end of the class
- Find your name or add it to one of the sheets
- Fill in 1 "Dot" per engagement
- More dots means more Overall Bonus Credit
- Lecture / Discussion sections will combine participation for this score, maxes out around 2% of overall grade

$$Bonus = \log_2(1 + LecDots + DiscDots)/2.5$$

Course Web Site

https://bakalian.cs.umd.edu/330

- Links for Syllabus, Staff Contact, Official Schedule
- Links to Project descriptions, intial code repos, etc

https://kauffman77.github.io/330/

- 4xx Lecture Materials
- Slides and Code and other goodies

Expectations

Kauffman can

- Provide guidance, entertainment, information, challenge
- Will do all of those in lecture, office hours, assignments, exams

Kauffman cannot

- Force you to pay attention, do your HW, attend classes, ask when you don't know, practice, learn.
- Cannot force you to **CARE**, the critical factor in any endeavor.
- Caring leads to effort. Effort leads to improvement. Constant improvement leads to success.

Kauffman's Expectation

- You care at least a little bit and will cultivate an attitude of curiosity and engagement
- You will put some effort into our time together as I have

Don't Give Up, Stay Determined!

Students have different experience levels. Some have lots and make things look easy. For others, everything is new and intimidating. No one knows all of this stuff. Everyone struggles at some point. Get help from the staff. Support each other. Your peers will remember when you help them move forward and when you try to hold them back.

Respect and learn from one another.

More DOTS Now!

- If you gave an answer during lecture or asked an interesting question. . .
- Come to the front of the room
- Add your name
- Give yourself a Dot
- Come back next time for more Dots