

Programming Languages

Victory Lap

CAR/CDR Pronunciation Guide

Function	Pronunciation	Alternate Name
CAR	<i>kar</i>	FIRST
CDR	<i>cou-der</i>	REST
CAAR	<i>ka-ar</i>	
CADR	<i>kae-der</i>	SECOND
CDAR	<i>cou-dar</i>	
CDDR	<i>cou-dih-der</i>	
CAAAR	<i>ka-a-ar</i>	
CAADR	<i>ka-ae-der</i>	
CADAR	<i>ka-dar</i>	
CADDR	<i>ka-dih-der</i>	THIRD
CDAAR	<i>cou-da-ar</i>	
CDADR	<i>cou-dae-der</i>	
CDDAR	<i>cou-dih-dar</i>	
CDDDR	<i>cou-did-dih-der</i>	
CADDDR	<i>ka-dih-dih-der</i>	FOURTH

and so on

Final Exam

- **Wednesday, December 16, 5:30PM**
- Material will be split somewhat evenly between pre-midterm and post-midterm
 - Including topics on homeworks and not on homeworks
- You will need to write code (Java, Racket) and English

Final Exam

Topics will be a subset of the following:

- All the stuff from the midterm (Racket in general, closures, recursion/tail-recursion, no mutation, lexical/dynamic scoping)
- Delayed evaluation, thunks
- Streams
- Memoization
- More in-depth static/dynamic typing
- More advanced OO concepts (e.g., circle-ellipse problem, late binding)
- Threading
- Interpreters and compilers
- Event-driven programming

Keys to the game:
Know what a topic is,
what it's good for, what it's
bad for, how to use it,
how it relates to other
topics, and how to code it.

Victory Lap

A victory lap is an extra trip around the track

- By the exhausted victors (us) 😊

Review course goals

- See if we met them

Some big themes and perspectives

- Stuff for five years from now more than for the final



Thank you!

- You all made this a great class
 - Great attitude about a very different view of software
 - Good class attendance and questions
 - Occasionally laughed at stuff 😊

Thank you!

- My second time teaching this course; not my area of expertise. (But I had a great time!)
- Feedback is appreciated on projects, tests, and their respective difficulty (too hard, too easy, just right?)



[From Lecture 1]

We have 14 weeks to learn *the fundamental concepts* of programming languages

With hard work, patience, and an open mind, this course makes you a much better programmer

- Even in languages we won't use
- Learn the core ideas around which *every* language is built, despite countless surface-level differences and variations
- *Poor* course summary: “We learned Racket and Java”

Today's class:

- Course mechanics
- *[A rain-check on motivation]*
- Dive into Racket

[From Lecture 1]

- Many essential concepts relevant in any programming language
 - And how these pieces fit together
- Use Racket and Java (possibly others) because:
 - They let many of the concepts “shine”
 - Using multiple languages shows how the same concept can “look different” or actually be slightly different
- A big focus on *functional programming*
 - Not using *mutation* (assignment statements) (!)
 - Using *first-class functions* (can’t explain that yet)

[From Lecture 1]

Learning to think about software in this “PL” way will make you a better programmer even if/when you go back to old ways

It will also give you the mental tools and experience you need for a lifetime of confidently picking up new languages and ideas

[From motivation lecture]

- A good mechanic might have a specialty, but also understands how “cars” (not 2014 Honda Civics) work
 - And that the syntax, I mean upholstery color, isn’t essential
- A good mechanical engineer really knows how cars work, how to get the most out of them, and how to design better ones
- To learn how cars work, it may make sense to start with a classic design rather than the latest model
 - A popular car may not be a good car for learning how cars work

[From motivation lecture]

This course focuses as much as it can on semantics and idioms

- Correct reasoning about programs, interfaces, and interpreters or compilers *requires* a precise knowledge of semantics
 - Not “I feel that conditional expressions might work like this”
 - Not “I like curly braces more than parentheses”
 - Much of software development is designing precise interfaces; what a PL means is a *really* good example
- Idioms make you a better programmer
 - Best to see in multiple settings, including where they shine
 - See PL X in a clearer light even if I never show you X

[From motivation lecture]

- No such thing as a “best” PL
- There are good general design principles for PLs
- A good language is a relevant, crisp interface for writing software
- Software leaders should know PL semantics and idioms
- Learning PLs is not about syntactic tricks for small programs
- Functional languages have been on the leading edge for decades
 - Ideas get absorbed by the mainstream, but very slowly
 - Meanwhile, use the ideas to be a better programmer in C++ and Python.

Benefits of No Mutation

1. Can freely alias or copy values/objects.
2. No need to make local copies of data.
3. No Circle-Ellipse problem.

Allowing mutation is appropriate when you are modeling a phenomenon that is inherently state-based (meaning there are variables that hold the "state" of the system and will need to change.)

- Performing an accumulation over a collection (e.g., summing a list) isn't!

Some other highlights

- Function closures are *really* powerful and convenient...
 - ... and implementing them is not magic
- Dynamic dispatch for OO really convenient...
 - ... and implementing that isn't magic either
- Static typing (and static checking) prevents certain errors...
 - ... but makes some types of code more complicated
- Multi-threading can make really neat programs...
 - ... but introduces a lot of sticky situations (synch, wait/notifyAll)
 - ... partially addressed by event-driven programming

From the syllabus

[Caveat: I wrote the goals, so not surprising I hope we met them.]

Successful course participants will:

- obtain an accurate understanding of what functional and object-oriented programs mean,
- develop the skills necessary to learn new programming languages quickly,
- master specific language concepts such that they can recognize them in strange guises,
- learn to evaluate the power and elegance of programming languages and their constructs,
- attain reasonable proficiency in a number of popular programming languages, and,
- become more proficient in languages they already know

From the "so-called experts" 😊

- Once a decade or so, ACM/IEEE updates a "standard CS curriculum"
 - A specification of what every CS undergraduate degree should teach its students
- Last updated in 2013!
 - Let's take a look at a draft and see how well we did.
 - (Note that not everything in the PL section of the draft will be taught in a single course.)

PL. Programming Languages (8 Core-Tier1 hours, 20 Core-Tier2 hours)

	Core-Tier1 hours	Core-Tier2 hours	Includes Electives
PL/Object-Oriented Programming	4	6	N
PL/Functional Programming	3	4	N
PL/Event-Driven and Reactive Programming		2	N
PL/Basic Type Systems	1	4	N
PL/Program Representation		1	N
PL/Language Translation and Execution		3	N
PL/Syntax Analysis			Y
PL/Compiler Semantic Analysis			Y
PL/Code Generation			Y
PL/Runtime Systems			Y
PL/Static Analysis			Y
PL/Advanced Programming Constructs			Y
PL/Concurrency and Parallelism			Y
PL/Type Systems			Y
PL/Formal Semantics			Y
PL/Language Pragmatics			Y
PL/Logic Programming			Y

What next?

- Take these ideas and use them in practice!
 - (But only where it makes sense)
- Be confident when reading documentation, unfamiliar code, learning a {new PL, new PL library, new programming paradigm}.
- Learn more Java!
- Stay in touch
 - Tell me when this class helps you out with something cool (seriously)
 - Ask me cool PL questions (may not always know the answer, but I can tell you where to find it)
 - Don't be a stranger: let me know how the rest of your time at Rhodes (and beyond!) goes... I really do like to know

