Course Website:

https://kmicinski.com/cis352-f23

CIS352 –
Principles of Programming Languages
Fall 2023

Instructor: Kris Micinski

We use writing to help ourselves structure our thoughts—revising, editing, restarting along the way

This class examines the process of writing and understand programs using a *systematic, iterative approach*

Want to learn "how to think" about programming

Why study programming languages?

- Learning a programming “language” is superficial
  - We want to learn **how to program in a specific paradigm**

Why study programming languages?

- Learning a programming “language” is superficial
  - We want to learn **how to program in a specific paradigm**

- Learning Python helps you a bit, but doesn’t directly enable ML
  - Need PyTorch / … for that!
- Learning C++ can help you write very fast code
  - But doesn’t teach you how to write fast, concurrent programs

- After you leave the class you will work in Java/Python/…, but you will almost certainly see overarching themes:
  - Mutability
  - Scope / Environments / …
  - Closures / Objects / …
  - Control-Flow / Tail-calls / loops / …
  - Exceptions / continuations / effects / monads / …
- These topics are the tricky parts of day-to-day programming

Languages are ~= APIs

After this course, you will likely never write a production programming language

But you will almost certainly build an API for something

And even now you use, learn, and think about APIs

Programming languages = APIs + syntax

- Parsing interesting; but orthogonal to our interest
- Instead, we will teach core principles for building languages:
  - Functions
  - Control-flow
  - Interpreters
  - Compilers
  - …

Course Objective

The main goal of this course is to teach you to **write completely correct code** that you can clearly explain and easily understand

We do this through **four coding projects**

Roughly (+/-) **5** programming **exercises**

Two written midterms

Course Goal

Course goal: *help you become an expert programmer*

How do we do that? **Focused, directed practice** at programming with rapid, repeated feedback will help you build intuition for patterns.

Course Goal

Course goal: *help you develop debugging intuition*

Via **Challenging projects** that **require** you to learn how to debug them due to their complexity.

Course Goal

Course goal: *learn to build good APIs*

By **implementing** key building blocks for programming language features

Logistics

In previous semesters I have used the **flipped classroom** style

This semester I will continue that, though I will recap the material in class. Think of posted video lectures as “the book.”

We will use Slack this semester

`https://kmicinski.com/cis352-f23`

Instructors

Kris Micinski (4th year asst. prof here @ SU)

Kris office hours: 30min after class Tu/Th (I leave @ 4:30)

I will have **debugging-oriented office hours** Wednesdays 2-3:30

Please avoid asking debugging-related questions after class

Feel free to write on Slack, but also make your own efforts

TA — Jialin Ye

Monday TA/lab hours: 9:30-12:30 (Room TBA next week)

Possible Friday OH upon request

Syllabus

Most up-to-date syllabus always available at:

https://kmicinski.com/cis352-f23/syllabus

Grading

- 50% Projects — 4 projects, each worth 12.5%
  - **Projects are the focus of the course**
- 10% Programming exercises — equally weighted
- 2 comprehensive midterms
  - Partly after midterm, and last day of class
- Can “revise” any incorrect answers (of *attempted* problems) for 50% points back on **first** midterm (not second); must be a “good effort attempt” (judged by me)
- The only students who have ever failed turned in <3 projects

Projects

This course has projects (with **deadlines**) that are assigned and graded via an **autograder**

https://autograder.org

You are expected to use the **Git interface** to the autograder; Autograder credentials will be sent out by the **first week**

Academic Integrity

No collaboration on code is allowed for projects—don't send / show / … anyone your code. Don't **post** any project code > 3 lines

The autograder employs elaborate measures that compare code (syntactically and semantically) to identify potential collaboration, then TAs and I check manually

"Hard coding" answers (for projects, i.e., recognizing specific inputs and providing correct outputs) is also an AI violation

I have reported roughly 25 cases over the last 5 years—all have been upheld; I will only report if I am sure there was copying

We **try** to make projects sync up with the material presented at the corresponding time in the course

Biggest indicator of success in the course is whether students are on-track with projects—try to never get behind; it becomes hard to catch up.

Project Grading

- Each project is graded on a percent scale; your grade is the % of tests that pass (18/20 tests passing = 90%)
- Projects always due at 11:59PM Syracuse time
- Projects up to 72 hours after deadline—15% penalty (max 85%)
- Projects up to end of course—25% penalty
  - I.e., you *can, in principle, always* get a 75%

Exams

- There will be a **two midterms** (second is a “final”)
- Both will be **in-class** and **written**
- Allowed one letter-sized (**single** sided) note sheet
- You may perform **corrections** for 1/2 marks back (first midterm)
  - More detail about these after first midterm
- I will release a practice midterm with the same question titles several days before both midterms; we will work it in class

Course FAQs

Q: Why teach Racket and not C++ / Java / JS / Rust / …

Everyone will have their own opinion on what language to use for a CS course—I realize that, and chose Racket for this course.

Racket is the language that allows you to write the **most direct implementation** of the projects we do in this course. If we used Haskell, Python, … the implementations would be doable—but would require much more time.

A goal of the course will be to teach you to use what we learn in whatever language you use (JS, C++, …)—we will teach features from other languages where possible.

Course FAQs

Q: Why emphasize functional programming / disallow `set!`

A: Functional programming is **simpler** (i.e., **more restrictive**), and thus easier to reason about. We will discuss how to implement state later on in the course, but we start by forcing students to program in a restricted purely-functional model because there are fewer opportunities for mistakes

Course FAQs

Q: Why projects? Why not small homeworks?
A: The bulk of the course, in practice, is doing the projects. This is reflected in the grade: exams are only 30%. Compared to courses that have homework requiring 5-20 line programs, our goal is to force you to program at a level where you can write ~100 lines of well-thought-out code doing something useful.

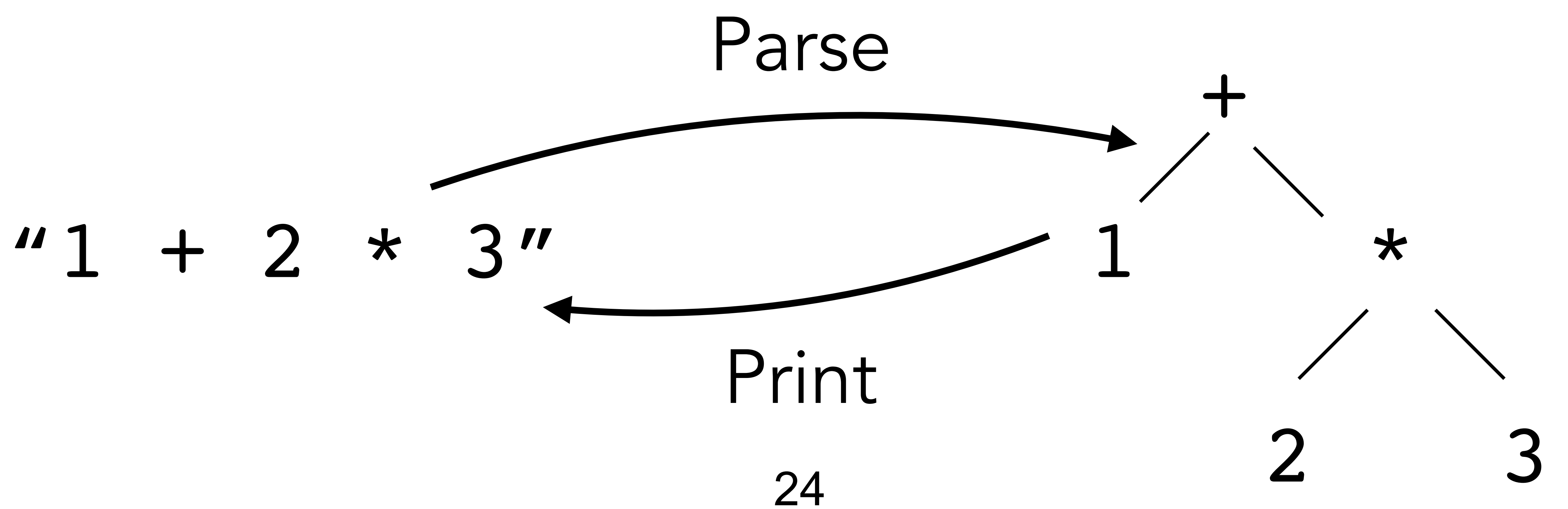
Syntax

A language's physical form, its identifiers and grammatical structure, is called its **syntax**

When we talk about programs, we often represent them as an **abstract** representation (e.g., an "abstract-syntax tree")

Tokenization and parsing is the task of turning raw syntax (stream of tokens) into an abstract representation

We will not cover parsing much



Semantics

PLs are unlike natural language—we *need* them to have a *precise, unambiguous* meaning

PLs have some systematically-defined meaning (semantics)

This can take several forms:

- Reference interpreter / compiler
- Written specification
- Machine-checked formal proof

Semantics

In this class we will mainly learn about semantics by building **interpreters**, though we will also speak of other kinds of semantics (e.g., the static semantics of type theory)

That’s enough course overview—let’s get into writing some Racket code.

Racket Basics

CIS352

Kris Micinski

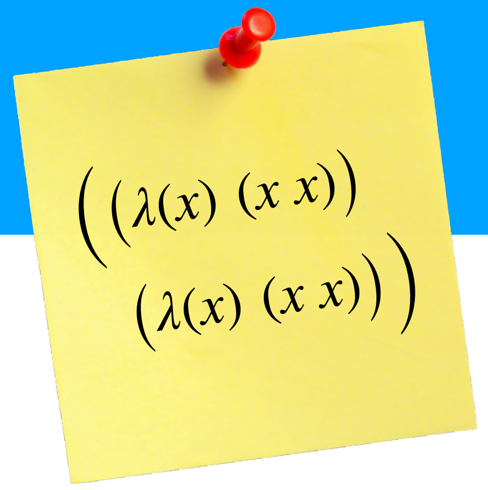
Racket

- **Dynamically-Typed:** variables are untyped, values typed
- **Functional:** Racket emphasizes functional style
  - Compositional—emphasizes black-box components
  - Immutability—requires automatic memory management
- **Imperative:** allows data to be modified, in carefully-considered cases, but doesn't emphasize "impure" code

Racket

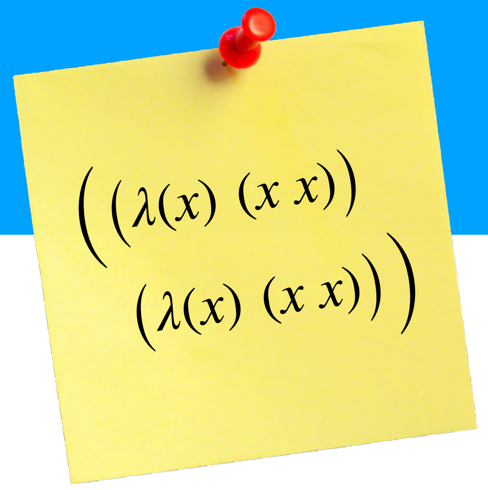
- **Object-oriented**: Racket has a powerful object system
- **Language-oriented**: Racket is really a language toolkit
- **Homoiconic:** the same structure used to represent **data** (lists) is also used to represent **code**

Example



Calculating the slope of a line in Racket

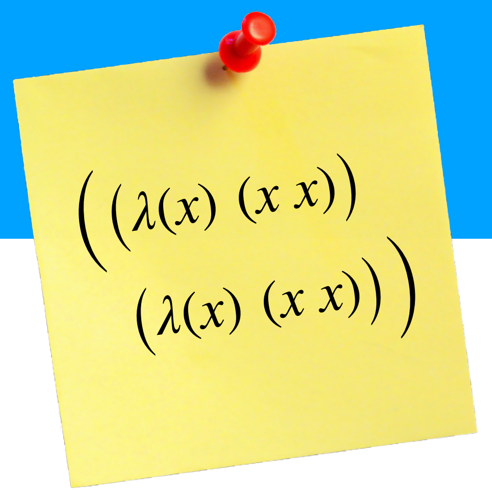
```
(define (calculuate-slope x0 y0 x1 y1)
  (/ (- y1 y0) (- x1 x0)))
```




```
(define (calculuate-slope x0 y0 x1 y1)
  (/ (- y1 y0) (- x1 x0)))
```

Prefix notation

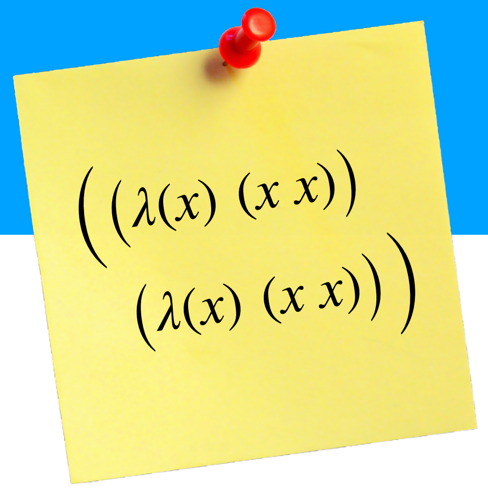
Example



Functions defined via prefix notation, too

```
(define (calculuate-slope x0 y0 x1 y1)
  (/ (- y1 y0) (- x1 x0)))
```

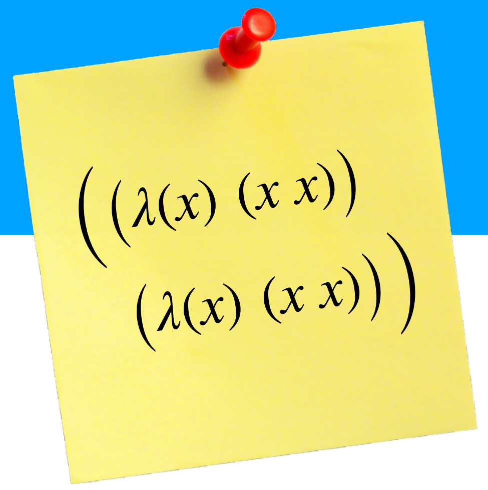
Example



Calls to user-defined functions also in prefix notation

```
(define (calculate-slope x0 y0 x1 y1)
  (/ (- y1 y0) (- x1 x0)))
// C - calculate-slope(0,0,3,2);
(calculate-slope 0 0 3 2)
```

Example



Note: preferred style puts closing parens at end of blocks

```
(define (calculuate-slope x0 y0 x1 y1)
  (/ (- y1 y0) (- x1 x0)))

(calculate-slope 0 0 3 2)
```

Basic Types

- **Numeric tower**. Numeric types gracefully degrade
  - E.g., `(* (/ 8 3) 2+1i)` is `16/3+8/3i`
  - Note that `2+1i` is a **literal** value, as is `2.3`
- **Strings** and **characters** (“`foo`” and `#\a`)
- **Booleans** (`#t` and `#f`) including logical operator (e.g., `or`)
  - Note that operators “short circuit”

Basic Types contd.

- **Symbols** are interned strings `'foo`
  - Implicitly only one copy of each, unlike (say) strings
  - Impact on space / memory usage
- The `#<void>` value (produced by `(void)`)

Exercise 

Compute the sum of the following:

- 2/3 and 1.5
- 3+8i and 3i
- 0 and positive infinity (`+inf.0`)

Exercise

Compute the sum of the following:

- **`(+ 2/3 1.5)`**

  `2.1666666666666665` (N.B., result is **inexact)**

- **`(+ 3+8i 0+3i)`**

  `3+11i`

- **`(+ 0 +inf.0)`**

  `+inf.0`