Course Website:

https://kmicinski.com/cis352-s26

CIS352 – Principles of Programming Languages Spring 2026

Instructor: Kris Micinski

TAs: Naveen Ashok, Jin Yang

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**
- Study the roots of programming languages to build a mental model of big ideas that unify different paradigms and languages
- Adapt our skills to new languages as new languages appear

Attendance Quizzes

Starting the second week of the semester, we'll start having regular (once every day or two) attendance quizzes. Each quiz is worth .5%, and you can get at most 5%. There will be at least 10 quizzes (probably more like 15-18). Quizzes will be given randomly roughly once a week during lecture.

Quizzes will be conducted via online portal (Socrative), bring a laptop/phone to class. You can also turn in on paper.

Instructors

Kris Micinski (4th year asst. prof here @ SU)
Jin Yang and Naveen Ashok (grad students at SU)

Kris office hours:
- One hour after class on Tuesday/Thursdays
- 1-2 hours on Wednesdays by appointment (email:)

Labs, TA office hours
- Monday labs
- TAs will also have several longer slots (on website / blackboard)

kkmicins@syr.edu

I will use email for most course communication. Please keep all discussion of course records to email rather than any unofficial channels.

Monday Labs

Monday labs are optional. There ***may*** sometimes be bonus exercises specific to the lab, but labs will generally be used for TA office hours, help with projects, etc.

Syllabus

Most up-to-date syllabus always available at:

https://kmicinski.com/cis352-s26/syllabus

Grading

- 60% Exams
  - Three midterms, one final, lowest exam drops
- 30% Programming homeworks
  - AI usage is permitted on these to an arbitrary degree
- 5% Written homework
  - 2-4 written assignments throughout the term
- 5% Participation Quizzes

Homework

This course has homework assignments (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**

Prepping your Computer for the Course

Steps you should take right now…

- Identify a terminal application you are comfortable using, you will need to use the command line for this class.
  - If you don't know it, this is an explicit skill we require you learn
  - At least the small amount of knowledge you need to run the grading scripts and submit projects via Git
- Download Dr. Racket
  - Go to "Help -> Configure Racket for Command Line"
- Install git on your machine
- Make sure Python3 is installed on your machine

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

Generative AI Policy

AI programming assistants have become radically more powerful in the past few years, clearly beating humans at programming speed for most tasks.

The goal of this class is to write a **small** amount of **really tricky, complex** code which has a **formally-articulated semantics.** Thus, using an AI to produce a large amount of code is often antithetical to the work we will do in this class.

However, it is a reality that generative AI will be a huge part of day-to-day programming for many workloads, and it will speed up software development (all things being equal) for most of its users.

Thus, **unrestricted AI use is allowed** on homework.

Generative AI Policy contd…

Must provide log of your chats (see syllabus!) for academic integrity purposes!

I particularly recommend Claude Code, which offers a rich TUI interface—other applications (Codex, etc.) are available.

Still very important to be in the "driver's seat," conceptually; what can you to do to avoid being checked out as you use the AI?

More here… https://kmicinski.com/claude-code-and-why-study-cs

Start **early** on homework

We **try** to make homework programming 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.

Homework Grading

- Each coding assignment 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 **three midterms (80min)** and a **final (120min)**
- Both midterms **in-class** and **written**
- Allowed one letter-sized (**double-sided**) note sheet
- I will release a practice midterm with the same question titles several days before both midterms; we will work it in class
- Final exam is slightly longer, but replaces the lower of your two midterm grades
- Each exam 20%, lowest drops (60% total from all exams)
- Skip the final if you believe it will not change your grade

Participation

- Participation quizzes on Socrative
- Quizzes weighted equally
- You can skip up to 30% of quizzes and still get 100%
- See syllabus for details…

Racket Basics

CIS352

Kris Micinski

Why learn Scheme/Racket?

- Our goal: rigorously study the ***semantics*** of programming languages.
- I will teach Racket, a language which is spiritually quite closely related to the untyped λ-calculus (a key concepts in computing and this class…)
- Racket's **syntax** is very, very simple—almost abrasively simple
- Racket's **semantics** can be defined in many ways, most of the semester we will focus on explaining *operational semantics* (i.e., "build an interpreter")
  - We will teach the semantics intuitively at first, as we learn the language
  - Then, we will use Racket as a common language for exploring lots of other exciting language features…

Other Languages (later on…)

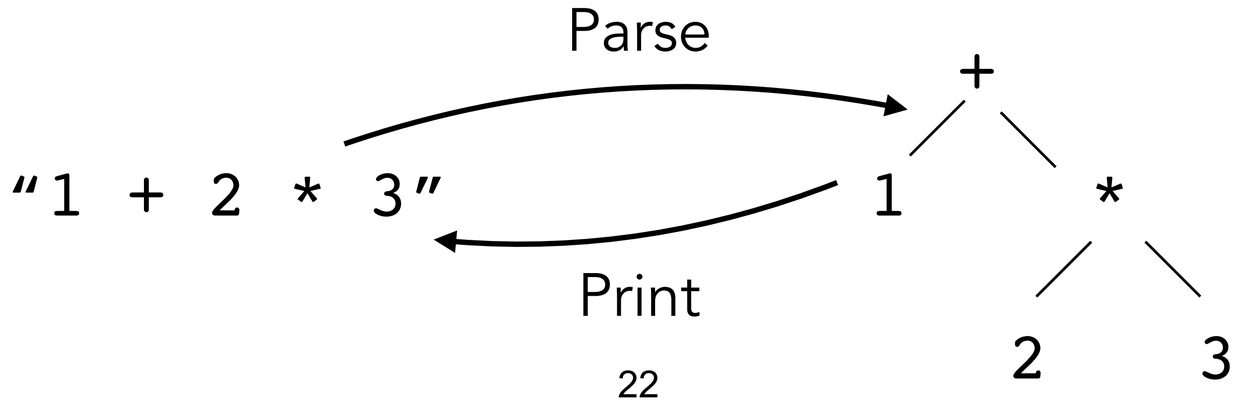
- I will also touch on other languages, especially Lean (formalizing mathematics) and Rust (low-level programming). You will not be required to know these languages for the purposes of exams.
  - I will also sometimes cover topics from C, C++, Java, JavaScript, and other common languages.

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



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

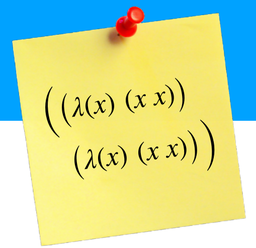
In this class we will mainly learn about semantics by building **interpreters**, though we will also speak of other kinds of semantics (e.g., the static semantics of type theory)

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

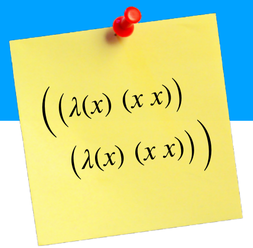
- **Object-oriented**: Racket has a powerful object system
- **Language-oriented**: Racket is really a language toolkit
- **Homoiconic:** the same structure used to represent **data** (lists) is also used to represent **code**



Calculating the slope of a line in Racket

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

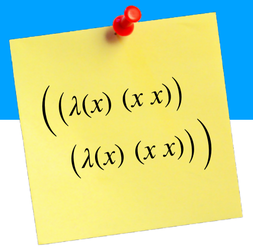
Example




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

Prefix notation

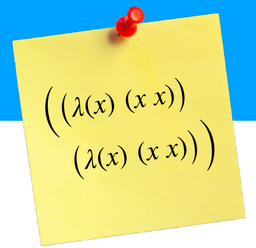
Example



Functions defined via prefix notation, too

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

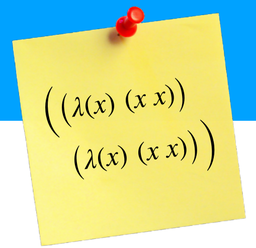
Example



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`