CS450
Structure of Higher Level Languages
Lecture 1: Course info, arithmetic in Racket
Tiago Cogumbeiro
About the course

- **Instructor:** Tiago (蒂亚戈) Cogumbreiro
- **Room:** M01-0409, McCormack
- **Schedule:** 3:00pm to 3:50pm, Monday, Wednesday, Friday
- **Office hours:** 1:00pm to 2:00pm, Monday, Wednesday, Friday

Course webpage

cogumbreiro.github.io/teaching/cs450/s20/
Syllabus

cogumbreiro.github.io/teaching/cs450/s20/syllabus.pdf

- Course divided into 8 modules
- 1 homework assignment per module
- Final grade: 90% homework + 10% participation
- **Homework grade:** average of 8 assignments (possibly weighted)
- **Participation grade:** in-class quizzes, attendance classroom/online, participation in forum
- **Classroom attendance is required!**

<table>
<thead>
<tr>
<th>Grade</th>
<th>Letter</th>
</tr>
</thead>
<tbody>
<tr>
<td>≥95</td>
<td>P</td>
</tr>
<tr>
<td>90 ≤</td>
<td>P</td>
</tr>
<tr>
<td>85 ≤</td>
<td>P</td>
</tr>
<tr>
<td>75 ≤</td>
<td>P</td>
</tr>
<tr>
<td>70 ≤</td>
<td>P</td>
</tr>
<tr>
<td>65 ≤</td>
<td>P</td>
</tr>
<tr>
<td>55 ≤</td>
<td>P</td>
</tr>
<tr>
<td>50 ≤</td>
<td>P</td>
</tr>
<tr>
<td>45 ≤</td>
<td>P</td>
</tr>
<tr>
<td>35 ≤</td>
<td>P</td>
</tr>
<tr>
<td>30 ≤</td>
<td>P</td>
</tr>
<tr>
<td>30 ≤</td>
<td>P</td>
</tr>
</tbody>
</table>
Course requirements

Checklist

- Install Racket 7.3: racket-lang.org
- Sign in on Gradescope: www.gradescope.com/courses/37850
- Sign in on Piazza: piazza.com/class/k5ubs34raz3ao
- Sign in on Estalee: www.estalee.com

Heads up

- Please, register using your UMB email address, otherwise you won't be able to submit your first homework.
- Homework 1 is due February 9 at 11:59pm and your homework assignment sheet must be picked up in person, as each student has a unique assignment. Please, contact me if you cannot pick up your homework assignment page in class.
Course overview
This course is **NOT**...

- **on algorithms**
  For a nice free book read *Algorithms* by Jeff Erickson.

- **an introduction on programming and computing**
  For a nice free book read *How to design programs* by Matthias Felleisen, Robert Bruce Findler, Matthew Flatt, Shriram Krishnamurthi

- **on programming with Racket**
  For a nice free book read *The Racket Guide* by Matthew Flatt, Robert Bruce Findler, and PLT
This course is...

- **on designing programming language features**
  We will focus mainly on functional and object-oriented programming.

- **on semi-formal specification**
  We will drive our course with precise mathematical notations and tests.

- **on programming patterns**
  We will characterize patterns and study abstractions of these patterns.

- **on purely functional programming**
  We will approach programming without using assignment (mutation).
Today we will learn

- a formalism to describe a programming language (Racket)
- the semantics of a programming language

How we will learn it

- We introduce one language feature at a time
  1. Syntax: We formalize each language feature (What)
  2. Example: We illustrate a feature with an example
  3. Semantics: We introduce how each language feature works (How)
Semantics

- **Abstract Syntax**: how we write something. Example, which characters/string we use to write a keyword, or a number.
- **Semantics**: what that something does/means (evaluation here means as the program runs)

In this class, we focus on the **semantics** of programming languages. We define the semantics of some programming language features.
1. We shall **not** print to output!
   Instead, we will use **assertions**.

2. We shall **not** mutate variables!
   Instead, we will use **persistent data structures**.

3. We shall **not** use loops!
   Instead, we will use **recursion**.
In Racket, **everything evaluates down to or is a value**. A Racket program consists of a preamble followed by zero or more expressions:

```
program = #lang racket expression*
```

1. Racket has no end-of-line delimiters (contrary to, say, C-like languages which use semi-colons)
2. Racket evaluates each expression from top-to-bottom, left-to-right

For space-constraint reasons, code listings might omit the preamble.

**Language specification**

- **Grayed out text** represents the concrete syntax
- **Italic text** represents a meta-variable
Expressions

Expressions can be values, among other things

expression = value | ⋮
Values

- **Numbers**
- Void
- Booleans
- Lists
- ...
Numbers
Numbers

All numbers are complex numbers. Some of them are real numbers, and all of the real numbers that can be represented are also rational numbers, except for \(+\text{inf.} \theta\) (positive infinity), \(+\text{inf.} f\) (single-precision variant), \(-\text{inf.} \theta\) (negative infinity), \(-\text{inf.} f\) (single-precision variant), \(+\text{nan.} \theta\) (not-a-number), and \(+\text{nan.} f\) (single-precision variant). Among the rational numbers, some are integers, because round applied to the number produces the same number.

Source: Racket Manual, Section 4.2
Hello, Numbers!

Your first Racket program

```racket
#lang racket
10 ; A positive number
+10 ; The plus sign is optional
-10 ; A negative number
0+1i ; A complex number
1/3 ; A rational number
0.33 ; A floating-point number

$ racket nums.rkt
10
10
-10
0+1i
1/3
0.33
```

Note: a semi-colon (;) initiates a comment section, which is ignored in Racket. A semi-colon is **not** a end-of-line marker, like in C-like languages.
Expressions are separated by white-space

These two programs are equal:

```racket
#lang racket
10 +10 -10 0 +1i 1/3 0.33
```

**Caveats:** -1 is different than `1` (notice the white space in between both characters). The former is the negative one, the latter is the expression `-` and the value 1. Similarly, `1/3` is a single rational number, whereas `1 / 3` are three expressions.
Function calls
Function call

Delimited by parenthesis and its constituents are separated by white-space characters. The first expression must evaluate to a function, the remaining expressions are the arguments. Each expression is evaluated to a value from left-to-right before applying the function.

```
expression = value | variable | function-call | ⋯
function-call = ( expression-func expression-arg* )
```

For instance, function call `(expt 2 3)`, for exponentiation, returns 2 raised to the power of 3. Function `sin` computes the sine function of its sole argument.

```
#lang racket
(expt 2 3)
(sin (expt 2 3))
```

$ racket nums-func.rkt
8
0.9893582466233818

**Note:** Function calls can be compounded, as the parameters of a function are arguments too.
No infix notation in Racket

There is **NO INFIX NOTATION** for arithmetic operations (unlike most languages). The usual arithmetic operations are all just variables: addition +, subtraction -, multiplication *, division /.

Example:

```
(* 3.14159 (* 10 10))
```

Note: In Racket parenthesis represent function application. Contrasted with most C-like languages where parenthesis in expressions are optional and only there to help the reader.