CS450

Structure of Higher Level Languages

Lecture 1: Course info, arithmetic in Racket

Tiago Cogumbreiro

About the course

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

Class structure

- **Live Q&A session Mondays, 3:00pm to 3:50pm via Zoom**
- **Pre-recorded videos available in YouTube,** around class time (3pm, Mo/We/Fr)

Support

- **Office hours** via direct messaging, video conferencing (Discord/Zoom)
- **Announcements** via direct messaging (Discord)
- **Forum/knowledge base** via issue tracker (Gitlab)

How we are doing remote teaching

- **Open door policy, via Discord.**
  - Message me at any time with your questions.
  - Channel questions answered first, direct-messages answered second.
  - I reply as soon as possible, during office hours in the latest.
- **Homework assignments** we use a grading server (Gradescope)
- **I record extra videos on demand**
  Please, don't be afraid to ask!

Course webpage

cogumbreiro.github.io/teaching/cs450/s21/

Syllabus

cogumbreiro.github.io/teaching/cs450/s21/syllabus.pdf

- Course divided into 8 modules
- 1 homework assignment per module
- Final grade: 95% homework + 5% participation
- ***Homework grade:*** average of **8 assignments** (possibly weighted)
- ***Participation grade:*** in-class quizzes, attendance classroom/online, participation in forum
- To get `D-` (`C-`) you need to have at least 7 assignments with `D-` (`C-`)
- **Monday** attendance is required!

	Grade		***Letter***
95 ≤	P		A
90 ≤	P	< 95	A-
85 ≤	P	< 90	B
75 ≤	P	< 85	B
70 ≤	P	< 75	B-
65 ≤	P	< 70	C+
55 ≤	P	< 65	C
50 ≤	P	< 55	C-
45 ≤	P	< 50	D+
35 ≤	P	< 45	D
30 ≤	P	< 35	D-
30 ≤	P		F

Course requirements

Checklist

- Install Racket 7.3: `racket-lang.org`
- Sign in on GitLab, comment on issue 1 (invitation by email)
- Sign in on Discord, say "Hi" in `#general` (invitation link in the GitLab page)
- Sign in on Gradescope, upload the template `hw1.rkt` (invitation by email)

Heads up

- Please, **register using your UMB email address**, otherwise you won't be able to submit your first homework.
- The deadline of homework assignment n is last class of module n plus 1 week

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 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***.

Your first program

Program

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-sentence 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 `+inf.0` (positive infinity), `+inf.f` (single-precision variant), `-inf.0` (negative infinity), `-inf.f` (single-precision variant), `+nan.0` (not-a-number), and `+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

```
#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:

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

```
#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))
| | |       | | |  |-> Number
| | |       | | |-> Number
| | |       | |-> Variable
| | |       |-> Function call
| | |-> Number
| |-> Variable
|-> Function call
```

> **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.

Evaluating a function call

Evaluation works from left-to-right from top-to-bottom

```
#racket lang
; Version 1:
(* 3.14159 (* 10 10))
; Version 2:
(* 3.14159 100)
;          ^^^- Evaluated (* 10 10)
; Version 3:
314.159
;^^^^^^- Evaluated (* 3.14159 * 100)
```