CS450

Structure of Higher Level Languages

Lecture 1: Course info, arithmetic in Racket, evaluation

Tiago Cogumbreiro
About the course

- **Instructor:** Tiago (蒂亚戈) Cogumbreiro (he/him)
- **Location:** (Y02-2300) Room 2300, 2 nd floor, University Hall
- **Schedule:** Monday, Wednesday / 5:30PM - 6:45PM

How to reach me

- **Office hours** via direct messaging (Discord), video conferencing (Zoom)
- **Announcements** in #cs450-news (Discord)
- **Q&A** in #cs450 (Discord)
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)

Course webpage

cogumbreiro.github.io/teaching/cs450/f22/
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
To get D- (C-) you need to have at least 7 assignments with D- (C-)
Academic dishonesty
Plagiarism in University

Copying code from others is wrong because:

- you do not learn
- you risk being expelled
- you are risking the other person being expelled
- you risk not completing your degree
- you risk being put on a list of cheaters (other universities may reject your application)
Plagiarism in the Industry

Is wrong, because:

- it is illegal
- you risk being dismissed from employment
- you risk being sued
Copying code (when it is right)

- software licenses define clear rules on how you can **copy, use, and change** other people's code
- open source promotes sharing of code
  - attribution is important (unless public domain)
  - good way to land on a job
**Plagiarism in CS 450**

- student's responsibility to learn the Student's code of conduct
- we use plagiarism detection *(renaming functions is not enough)*
- we compare against solutions from past years (and instructor)
- be careful when working with others, any sharing code may trigger
- the plagiarism detection tool can detect code sharing among students
Plagiarism in CS 450

Zero Tolerance

- statistically, there will be plagiarism this semester
- if I contact you regarding plagiarism, there will be zero tolerance:
  - You will get an F in this course
  - You will be reported to the university

If you need more time to complete an assignment, ASK
Course requirements
Course requirements

Checklist

- Install Racket 7.3: racket-lang.org
- Sign in on GitLab (invitation by email)
- Sign in on Discord, say "Hi" in #cs450-lounge (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
Why learn the Structure of Higher Level Languages?
I postponed this discussion, because I felt that you are now better suited to understand and relate to the points being made.

- Why learn the fundamental concepts in all programming languages?
- Why learn different languages?
- Why focus on functional programming?
- Why use Racket?

Disclaimer

- Most of these claims are opinions
- These will be mostly informal claims
- We are not trying to find the best language (or programming model)
Overview

- Languages are just tools, learn which language is amenable to what context
- The best programming language does not exist (theoretically most languages are equivalent)
- Different languages have different characteristics that favour different domains: for instance, functional languages being used in Programming Language research, C/Fortran in scientific/high-performance computing
- A programming language is a computing interface: it is crucial to understand its meaning
- The importance of first-class functions and avoiding mutation
Semantics and idioms

Why should we care about language semantics?

- A language is a *computing user interface*. We are learning reusable, cross-cutting patterns.
- The semantics must be *unambiguous and precise*. It is not a matter of personal opinion how a conditional expression works. Language features must be described unambiguously to users.
- The semantics defines a software contract. Is the bug in the client's bug, or is it in our code?
- Language idioms (patterns) are *transferrable knowledge*. Understanding idioms (patterns) teaches you something that can be applied across languages and technologies.
How are all languages similar?
How are all languages the same?

- **Theoretical:** Any input-output behavior implementable in language X is implementable in language Y (Church-Turing thesis), and *equivalent to the \( \lambda \)-calculus without numbers*
- **Practical:** Reoccurring fundamentals: variables, abstraction, recursive definitions
How are languages different?
Disclaimer

Languages are not slow/fast

- A language **implementation** is fast/slow, not the language itself
- Certain languages computational models are more amenable to implement efficiently
- Languages are user interfaces of computational models

How different languages behave in different contexts?
Why is C faster than all other languages?

Is it because C is "close to the metal?" That is, is C fast because its semantics matches the processor's semantics?
Why is C faster than all other languages?

Is it because C is "close to the metal?" That is, is C fast because its semantics matches the processor's semantics? **No!**

- Which processor? How could it match the semantics of all processors?
- Which compiler? The key of C's success lays in having good compilers.
- C compilers are fast because C is **old and its interface remains stable!**
- Popular C compilers are **really** good at optimizing the target language.
- There is a set of good practices to write optimizer-ready C code

Take away

The facts above make C quite successful in High Performance Computing (large scale scientific codes).

Source: C Is Not a Low-level Language: Your computer is not a fast PDP-11. David Chisnall. ACM Queue vol 16, no 2. 2018
Why is Python slow multithreading?

- CPython (the main implementation of Python) is conditioned by the GIL (the Global Interpreter Lock) which effectively serializes parallel execution.
- To parallelize code we must run multiple processes, where shared memory is especially slow, which, in turn, slows down compute-bound programs.

Take away

- Avoid running compute-bound parallel codes in Python. Maybe choose C?

We solve the equation $\text{SEND} + \text{MORE} = \text{MONEY}$ where each letter represents a digit in Prolog using a constraint language programming module:

```prolog
sendmore(Digits) :-
    Digits = [S, E, N, D, M, O, R, E],  % Create variables
    Digits ins 0..9,                 % Associate domains to variables
    S #\= 0,                         % Constraint: S must be different from 0
    M #\= 0,                         % Other constraints
    all_different(Digits),
    1000*S + 100*E + 10*N + D         % Other constraints
    + 1000*M + 100*O + 10*R + E
    #= 10000*M + 1000*O + 100*N + 10*E + Y,
    label(Digits).                   % Start the search
```

Take away

Some problems are more amenable to certain programming languages.
How are languages different?

1. **The implementation matters:** A language implementation may be conditioned (faster/slower) in certain contexts

2. **The model matters:** Certain problems are simpler/more efficient to write in specific languages

3. **The domain matters:** A technology your business needs may only be available in some language (say TensorFlow in Python)
Why learn different languages?

- Learn at least one new language every year.


Why should you care

- Deeper understanding of the differences and the similarities between languages
- Learn different approaches to the same problems
- More job opportunities
- Better technology choices (some technologies are only available in specific languages)
Why functional programming?
What is functional programming?

- Mutation is discouraged
- Higher-order functions serve as a generalization device

Why should we care?

- These features help designing correct, elegant, and efficient software
- Functional programming languages are heavily favoured by PL researchers, which means they serve as a test bed for PL design. Functional programming is close(r) to math formalism, thus implementation is usually simpler in functional programming languages.
- **Functional programming is trendy!** C++/Java/C#/Python/Javascript are all incorporating functional programming idioms.
Why should we discourage mutation?

- Simpler to reason about: no surprises passing a data-structure to functions/objects
- Concurrency-ready: read-only means no race conditions (and no locks), which leads to simpler, faster code

Who is using it?

- immutable.js for JavaScript by Facebook
- vavr, PCollections, the Scala runtime, and the Closure runtime for Java
- immer for C++
- immutable collections for .NET
Why should we use higher-order functions?

- Simpler interface than objects (which method? which order?)
- Can be combined effectively (frameworks on combining functions)
Most programming languages features started out in functional programming languages.

- Garbage collection (LISP, 1959)
- Higher-order functions (lambda expressions in C++, C#, Java, Python) introduced in LISP (1959) and in ISWIM (1966)
- Type inference, e.g., auto in C++, var in C# (Hindley-Milner-Damas)
- Algebraic-data types and pattern matching (1970s in Hope)
- Recursion
A new wave of languages

Many new interesting programming languages

- Swift: next-generation programming language for Apple systems
- Rust: functional programming meets system programming
- F#: an ML derivate for the .NET ecosystem
- Elixir: highly-available distributed system
- Clojure: a LISP-influenced language for the JVM and the web
How are we using functional programming?

- **OCaml**: web development (Facebook/Meta), distributed systems (Docker), finance (Tezos, Jane Street, Bloomberg, Aesthetic Integration), hardware virtualization (Citrix)
- **Haskell**: verification (Facebook), distributed systems (Google), compilers (Intel), distributed systems (Microsoft)
- **Erlang**: communication (WhatsApp), ads (AddRoll), web backend (Bet365), finance (Goldman Sachs)
- **Elixir**: spam prevention (Pinterest), micro services (Lonely Planet)
- **F#**: data analysis (Kaggle), trading (Credit Suisse), gaming backend (GameSys)
- **Racket**: game scripting (Naughty Dog), image processing (YouPatch)
- **Scala**: middleware (Twitter), database (Netflix), microservices (Tumblr), web (The Guardian)

Honorable mentions

- ReasonML, Elm, PureScript, ClojureScript
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
In Racket, **everything evaluates down to or is a value**. A Racket program consists of a preamble followed by zero or more expressions:

```racket
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

\[ \text{expression} = \text{value} \mid \cdots \]
Values

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

```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:

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

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

```racket
#racket lang
; Version 1:
(* 3.14159 (* 10 10))
; Version 2:
(* 3.14159 100)
; ~~~~~ Evaluated (* 10 10)
; Version 3:
314.159
;~~~~~~~~~ Evaluated (* 3.14159 * 100)
```
Arithmetic expressions example

\[ ((11 \cdot 15) + (14 + 4)) + \left( \frac{3}{9} - (14 \cdot 3) \right) \]
Arithmetic expressions example

\[
((11 \cdot 15) + (14 + 4)) + \left(\frac{3}{9} - (14 \cdot 3)\right)
\]
A longer example

\[
\begin{align*}
(+ & \quad (+ \\
(* & 11 \ 15) & \quad (+ \ 14 \ 4)) \\
(- & \quad (- \\
(/ & 3 \ 9) & \quad (/ \ 3 \ 9)) \\
(* & 14 \ 3))) & \quad (* \ 14 \ 3)))
\end{align*}
\]

\[
\begin{align*}
(+ & \quad (+ \\
165 & \quad (+ \ 14 \ 4)) \\
(- & \quad (- \\
(/ & 3 \ 9) & \quad (/ \ 3 \ 9)) \\
(* & 14 \ 3))) & \quad (* \ 14 \ 3)))
\end{align*}
\]

\[
\begin{align*}
(+ & \quad (+ \\
183 & \quad (+ \ 14 \ 3)) \\
(- & \quad (- \\
(/ & 3 \ 9) & \quad (/ \ 3 \ 9)) \\
(* & 14 \ 3))) & \quad (* \ 14 \ 3)))
\end{align*}
\]
A longer example

\[
\begin{align*}
&\left( + \right) \\
&\quad \left( + \right) \\
&\quad \left( \ast 11 15 \right) \\
&\quad \left( + 14 4 \right) \\
&\left( - \right) \\
&\left( / 3 9 \right) \\
&\left( \ast 14 3 \right) \\
&\left( - \right) \\
&\left( / 3 9 \right) \\
&\left( \ast 14 3 \right) \\
&\left( + \right) \\
&\quad \left( 165 \right) \\
&\quad \left( + 14 4 \right) \\
&\left( - \right) \\
&\left( / 3 9 \right) \\
&\left( \ast 14 3 \right) \\
&\left( + \right) \\
&\quad \left( 183 \right) \\
&\left( - \right) \\
&\quad \left( / 3 9 \right) \\
&\left( \ast 14 3 \right) \\
&\left( + \right) \\
&\quad \left( 183 \right) \\
&\quad \left( -125/3 \right) \\
&\left( + \right) \\
&\quad \left( 424/3 \right)
\end{align*}
\]
Interpreting an error in Racket

What would happen if we call a function using the infix notation?

\[(3 \div 9)\]
Interpreting an error in Racket

What would happen if we call a function using the infix notation?

```
(3 / 9)
```

; application: not a procedure;
; expected a procedure that can be applied to arguments
; given: 3
; [,bt for context]
Interpreting an error in Racket

What would happen if we call a function using the infix notation?

```
(3 / 9)
```

; application: not a procedure;
; expected a procedure that can be applied to arguments
; given: 3
; [,bt for context]

**Line 1**

The **subject** is application. Application is short for function application, aka **calling a function**.

The **symptom** is **not a procedure**.

Something that should be a procedure is not. Recall, procedure = **function**.
Interpreting an error in Racket

What would happen if we call a function using the infix notation?

```
(3 / 9)
```

; application: not a procedure;
; expected a procedure that can be applied to arguments
; given: 3
; [bt for context]

The **subject** is application. Application is short for function application, aka **calling a function**.

The **symptom** is not a procedure. Something that should be a procedure is not. Recall, procedure = **function**.

Calling a function requires a function, but we provided something else.
Interpreting an error in Racket

What would happen if we call a function using the infix notation?

```
(3 / 9)
```

; application: not a procedure;
; expected a procedure that can be applied to arguments
; given: 3
; [,bt for context]

### Line 1
The **subject** is application. Application is short for function application, aka **calling a function**.

The **symptom** is not a procedure. Something that should be a procedure is not. Recall, procedure = **function**.

### Line 2
Calling a function requires a function, but we provided something else.

### Line 3
We see what was given instead (number 3, rather than a function).
Is this example a legal Racket program?

```racket
#lang racket
sin
```
Is this example a legal Racket program?

```racket
#lang racket
sin
```

Yes! `sin` is a variable, so a valid expression. Hence, Racket just prints what is in variable `sin`.

```
$ racket sin.rkt
#<procedure:sin>
```

**Note:** In Racket lingo the word *procedure* is a synonym for function.
Racket specification

\[
\begin{align*}
\text{program} &= \text{#lang racket expression}* \\
\text{expression} &= \text{value} \mid \text{variable} \mid \text{function-call} \mid \cdots \\
\text{value} &= \text{number} \mid \cdots \\
\text{function-call} &= (\text{expression}+) 
\end{align*}
\]