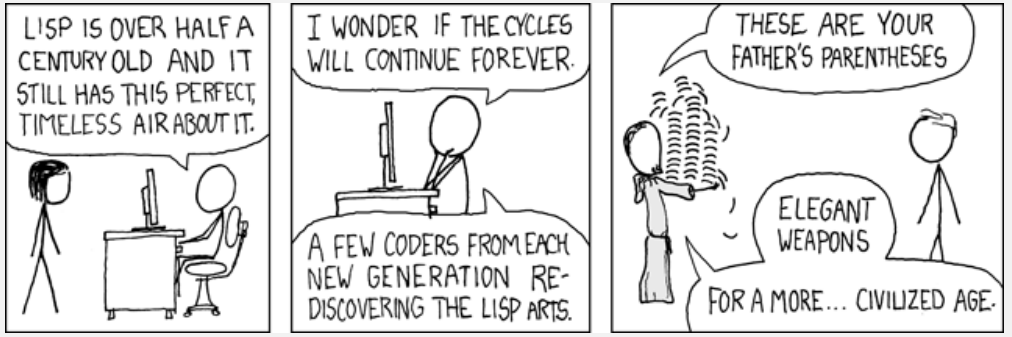
CS450
High Level Languages
UMass Boston Computer Science

Tuesday, February 3, 2026



Recursive Functions of Symbolic Expressions and Their Computation by Machine, Part I

John McCarthy, Massachusetts Institute of Technology, Cambridge.

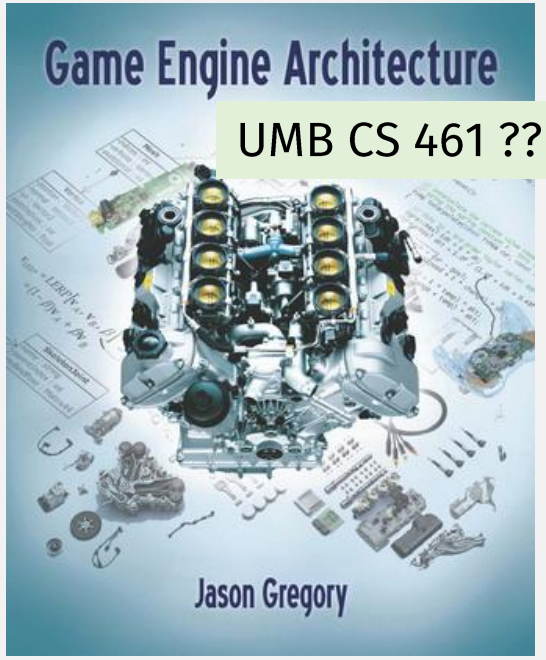
April 1960

`LISP` coming back into style..

1 Introduction

A programming system called LISP (for LISt Processor) has been developed for the IBM 704 computer by the Artificial Intelligence group at M.I.T. The system was designed to facilitate experiments with a proposed system called the Advice Taker, whereby a machine could be instructed to handle declarative as well as imperative sentences and could exhibit "common sense" in carrying out its instructions.

- Programs are **expressions** (not sequences of instructions!)
- S-expression syntax (parens, like Racket!)
  - "code is data, data is code"
  - `(list + 1 2)` is both a program and a list of "symbols"
- Invented: `if-then-else`, `lambda`, recursion, gc (no ptrs), `eval`



UMB CS 461 ??

LISP

(First "high-level" language)

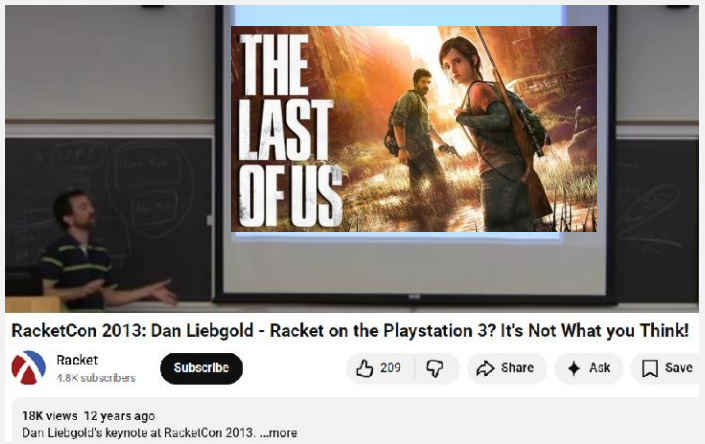
Clojure

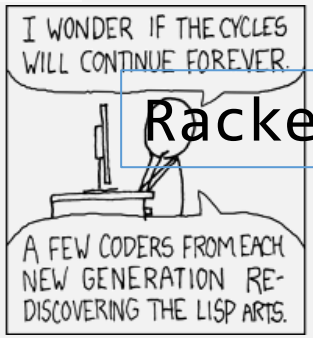
JVM

Scheme

Racket

Game Oriented Assembly Lisp (GOAL)	2000s	Andy Gavin	Video game programming language developed by Andy Gavin and the *Jak and Daxter* team at Naughty Dog; written using Allegro Common Lisp; used in developing the full game series





Racket450

450

Recursive Functions of Symbolic Expressions and Their Computation by Machine, Part I

John McCarthy, Massachusetts Institute of Technology, Cambridge.

April 1960

1 Introduction

A programming system called LISP (for LISt Processor) has been developed for the IBM 704 computer by the Artificial Intelligence group at M.I.T. The system was designed to facilitate experiments with a proposed system called the Advice Taker, whereby a machine could be instructed to handle declarative as well as imperative sentences and could exhibit "common sense" in carrying out its instructions.

LISP

(First "high-level" language)

Lisp is worth learning for the profound enlightenment experience you will have when you finally get it; that experience will make you a better programmer for the rest of your days, even if you never actually use

PAUL GRAHAM

BEATING THE AVERAGES

Want to start a startup? Get funded by Y Combinator.

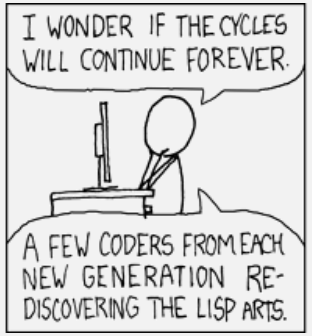
(This article is derived from a talk given at the 2001 Franz Developer Symposium.)

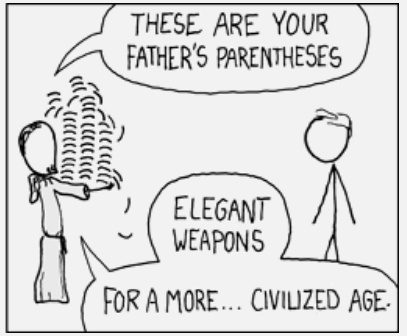
In the summer of 1995, my friend Robert Morris and I started a startup called Viaweb. Our plan was to write software that would let end users build online stores. What was novel about this software, at the time, was that it ran on our server, using ordinary Web pages as the interface.

yahoo! shopping ----> shopify

Another unusual thing about this software was that it was written primarily in a programming language called Lisp. It was one of the first big end-user applications to be written in Lisp, which up till then had been used mostly in universities and research labs.

hypothesis was that if we wrote our software in Lisp, we'd be able to get features done faster than our competitors, and also to do things in our software that they couldn't do. And because Lisp was so high-level, we wouldn't need a big development team, so our costs would be lower. If this were so, we could offer a better





The Other Side …

r/programmingcirclejerk • 2 yr. ago
LAUAR gofmt urself

Lisp is the worst family of languages in existence. Literally nothing can change my mind. Cool tool thingy, but you basically made a tool to teach blasphemy

It's a language that syntactically is awful. It sacrifices arbitrary elegance for practicality (human readability)

Racket does not compare to real, actual programming languages. In the real world, however, software developers use actual, practical languages like Python, C++ and to a lesser extent Javascript. The thought of using Racket never crosses their mutable variable-corrupted minds.

r/uwaterloo • 5 yr. ago
itsatrap12121

I HATE RACKET

UoT first year students are learning **python**

WHY DO WE HAVE TO DO FUCKING **RACKET** SHIT

RACKET IS GARBAGE

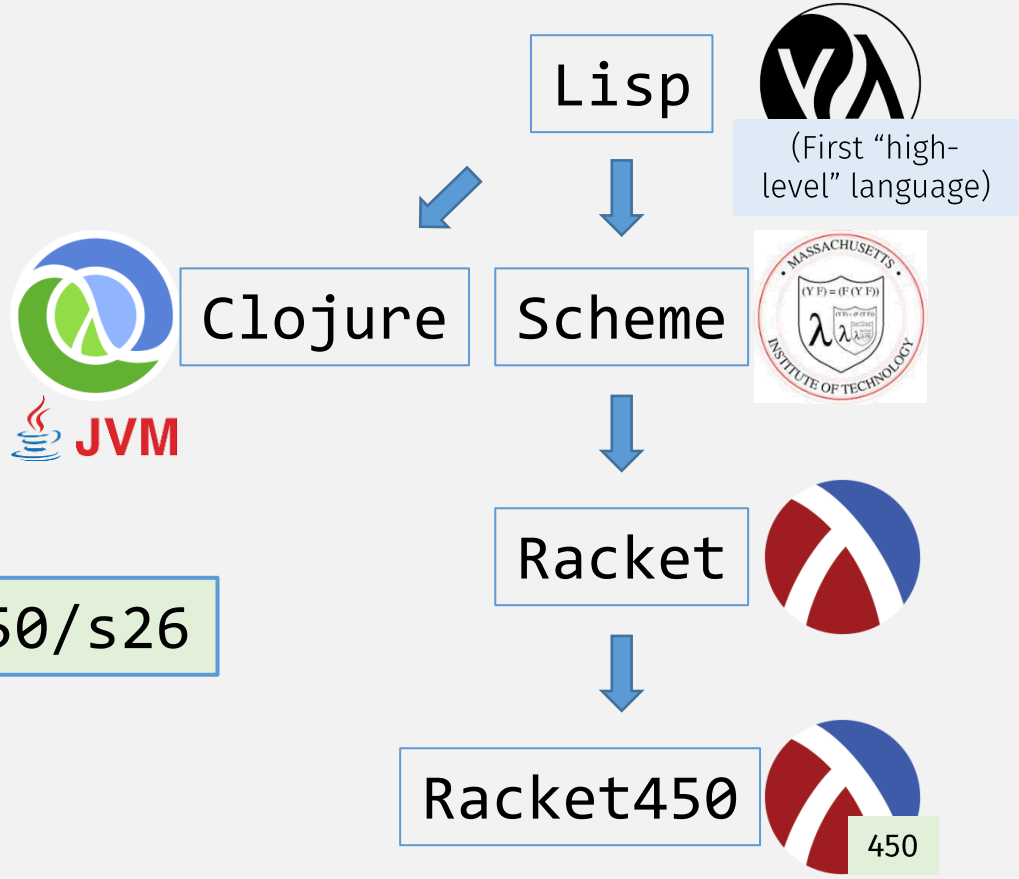
Fortunately … this **course** is <u>not</u> **about Lisp, Racket,** or **any other language.** It is **language-agnostic!**

It's about <u>general</u>, **high-level programming principles** … that can be used when programming in <u>any language</u>

Logistics

- HW 0 past due
  - ~~due: Tue 2/3 11am EST~~
  - Get it in ASAP!
- HW 1 out
  - due: Tue 2/10 11am EST
- Course web site:

  `https://www.cs.umb.edu/~stchang/cs450/s26`
- Add / drop ends Thurs 2/5
- If staying in the course:
  - confirm you read the course website and policies as part of hw1



Today's Focus: Being Ready For the Course

I don't know what _____ is …

… and at this point, I'm too afraid to ask!

(don't be this person)

Racket

450 edition

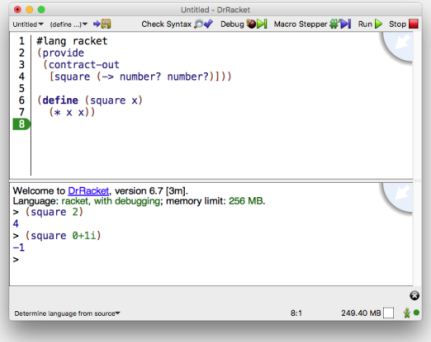
Web: `racket-lang.org`

Download: `download.racket-lang.org`

- Linux: `https://snapcraft.io/install/racket/ubuntu`

Version: **8.14+**

IDE: **DrRacket** (easiest)

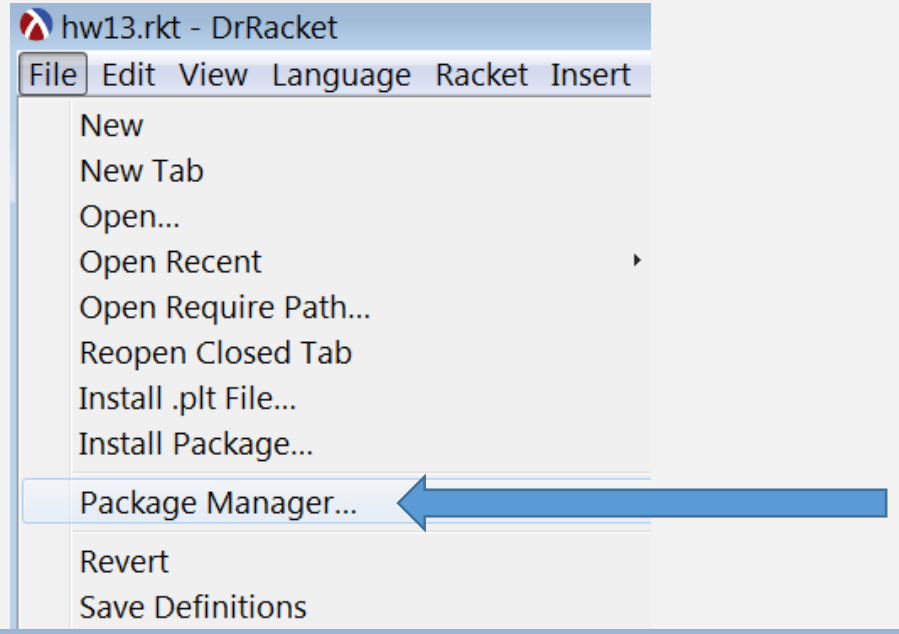


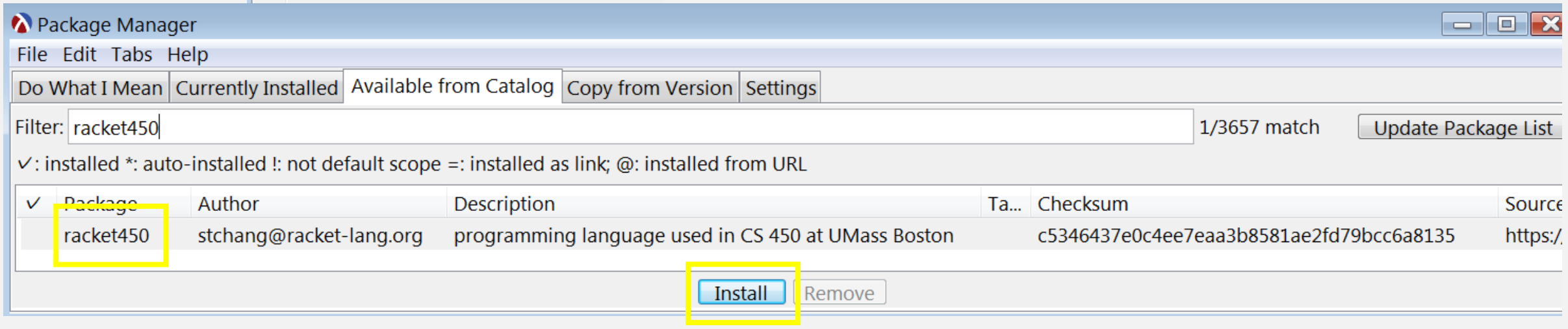
Docs: `docs.racket-lang.org`

forum:

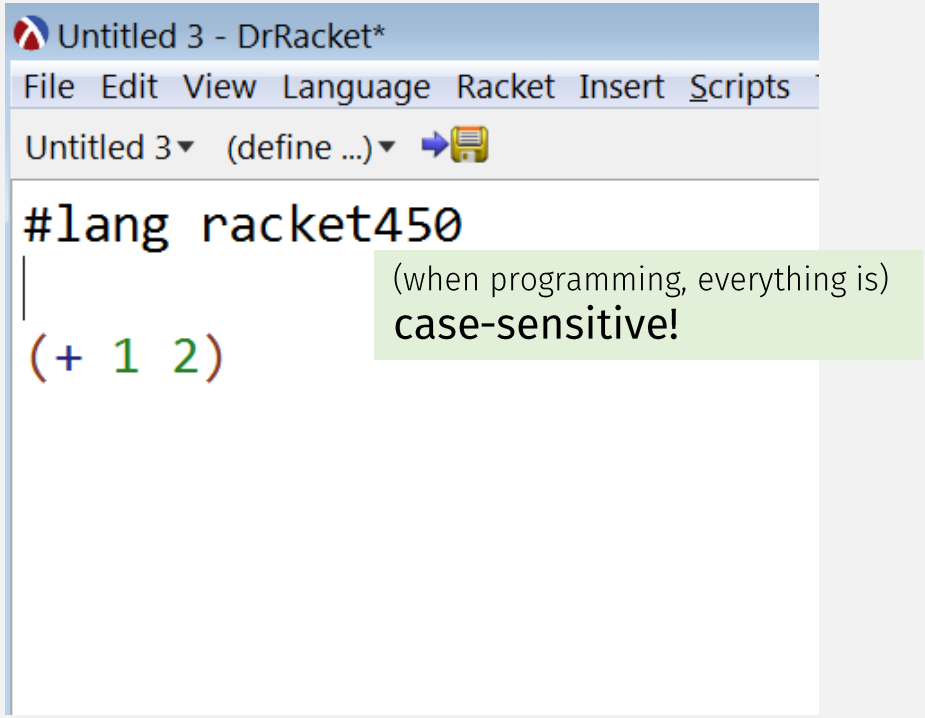
- HW help: **Piazza** (please don't email course staff)
- General: `racket.discourse.group`

Installing “racket450”





Using “racket450”



(textbook for this course)

How to Design Programs, 2nd ed.

Lessons:

- **How** to **“solve problems”,** i.e., **program**, **from scratch**

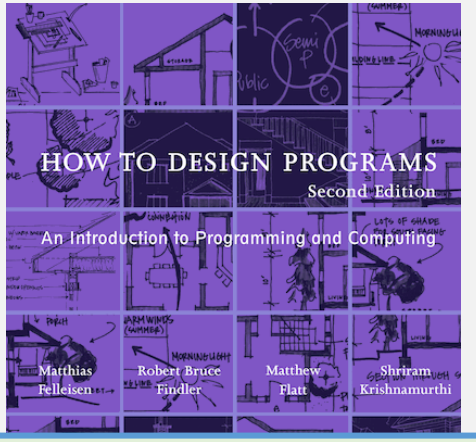
- **Programs** are (also) for **high-level communication**

I.e.,

- **Programs** are more than **“my code works”**
- … must be **readable / explainable by others!**

Available free at: **htdp.org**

- Can buy paper copy (make sure it’s 2nd ed)



This is our rulebook!

Every **org / company** has its own **rules for how to write clean, readable programs**

Git and Github

Should be done?

- Create `Github account`
  - And **tell me** (in survey – must log in with umb account)

- Install `git client`
  - GUI or command line ok

- Learn basic `git` commands?
  - `Clone, push, pull, fork, branch`

Other Logistics

- GitHub – store HW code here
- Piazza – ask HW questions here (don't email staff)
  - Faster response
  - Helps other students
  - HW posts should be public (anonymous ok)
- Gradescope – submit HW here
- Read course web page?

`https://www.cs.umb.edu/~stchang/cs450/s26`

Note: Autograders are for … Graders

- (A draft version) **may occasionally be released early**
  - … for your **benefit** … (or **detriment?**) – use at your own risk!
  - Not a debugging tool
- **Not guaranteed to be the same as final grader**
  - **Official grading is done after due date**
- **Not taking questions/feedback about autograder**
  - E.g., **“why does my code fail the autograder?”**
  - will be ignored
- *Repeat:* **Staff has no obligation to provide an autograder**
  - or **answer any questions about whether code is “correct”**

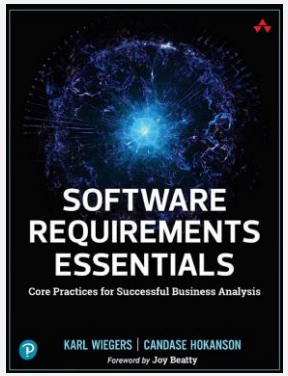
My goals for the course

1. **Teach students** about **high-level languages**

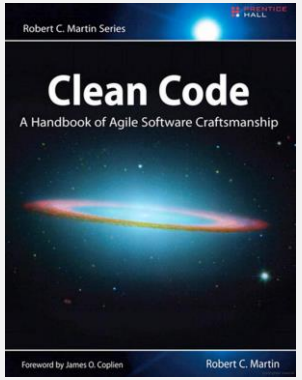
2. **Prepare students** with some **post-UMB CS career skills**
   - (No autograder in real-world … unless you write it!)

Interlude: Software Dev 101

(framework with which to view the course)

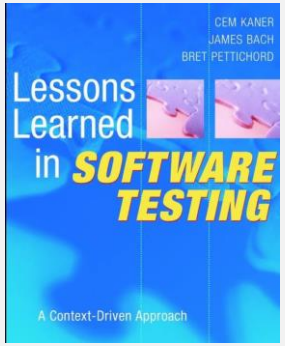


1. Write Specifications / Requirements
   - Figure out what the **program should do**



2. Implement code
   - Make the **program**

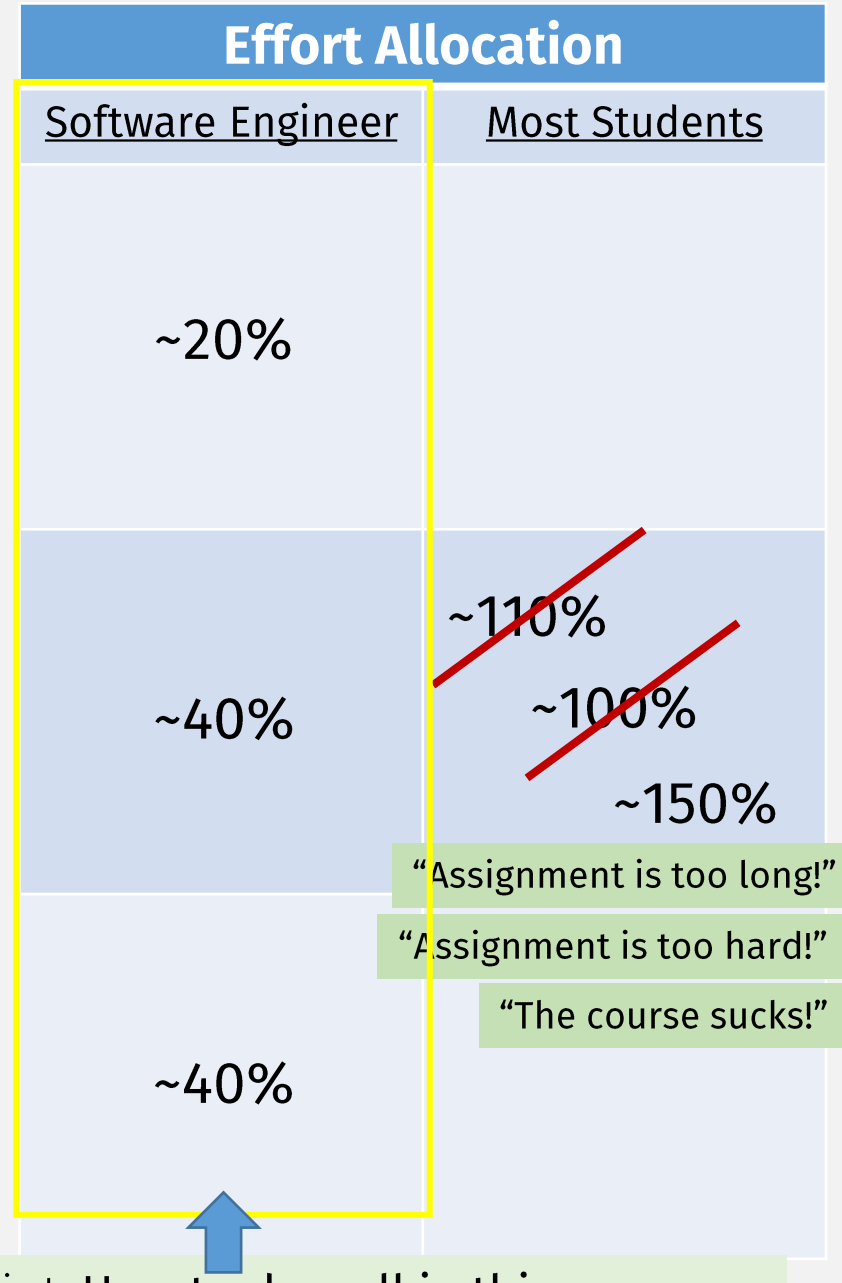
3. Verify correctness (i.e., **testing**):
   - Check the **program does what it should do?**



Interlude: Software Dev 101

1. Write Specifications / Requirements
   - Figure out what the **program should do**
2. Implement code
   - Make the **program**
3. Verify correctness (i.e., **testing**):
   - Check the **program does what it should do?**

(This is **not** a **course about** the details of **Software Dev.**
But we will use it as a **framework for the ideas taught**)



How to Effectively Ask Questions (that will get a useful answer)

1. Write Specifications / Requirements
   - Figure out what the **program should do**

Ask clarification questions about this first …

… with small examples

2. Implement code
   - Make the **program**

3. Verify correctness (i.e., **testing**):
   - Check the **program does what it should do?**

How to Ask Clarification Questions!

For every programming task …

- You will always be initially **unsure** about the **"correct behavior"**
- your **first task** is to **clarify this uncertainty** (don't want until after grading!)
  - i.e., **understand the problem!**

- **HW questions should include specifics / examples**, e.g.:
  - "I'm designing function "**p**" to have *nine* arguments." ("Am I understanding the problem correctly?")
  - "I expect **<example expression>** to evaluate to **<answer>**. "
  - "I expect **(my-grade "CS450" 2026)** to evaluate to **"A+"** "
    ("Am I understanding the problem correctly?")

- **Do not write any code until you understand the problem**
  - (Otherwise, how can you possibly come up with a correct solution?)

How to Ask Bad HW Questions (that will not get a useful answer)

1. Write Specifications / Requirements
   - Figure out what the **program should do**

2. Implement code
   - Make the **program**

Questions about code are pointless … if you **don't understand what the program is supposed to be doing**

3. Verify correctness (i.e., **testing**):
   - Check the **program does what it should do?**

Bad HW Questions (that will not get a useful answer)

- Not allowed: **Questions about code correctness** (they will be ignored)
  - E.g., "Is this code right?"
  - Instead: **Need to explain what you think "correct" means (with example)**

- Not allowed: **Vague questions**
  - E.g., "What's wrong???"
  - E.g., "How do I start this assignment???"
  - Instead: **Ask with specifics / examples**

- **Staff will not debug code** (if you don't understand it, no one else will)
  - E.g., "Help me find the problem somewhere in this 1000 line progam!"
  - Instead: **Using an example, remove code until the problematic line is found**

Racket (Very) Basics

- File extension: `.rkt`
- First line: `#lang racket450`
  - (The HtDP Textbook uses “Student Languages” but we will not use those)
  - (Differences will be explained)
- syntax: “S-expressions”
- Comments:
  - `;` line
  - `#;` S-expression
- Identifiers:
  - everything except: `( ) [ ] { } “ , ‘ ` ;`
  - And not: `|  \  #`
  - **Case sensitive!**

Racket Basics

This position must be an (arithmetic expression that evaluates to a) **function value**

- Function call: **prefix notation** (fn name first)
  - **Easier to write multi-arity functions**

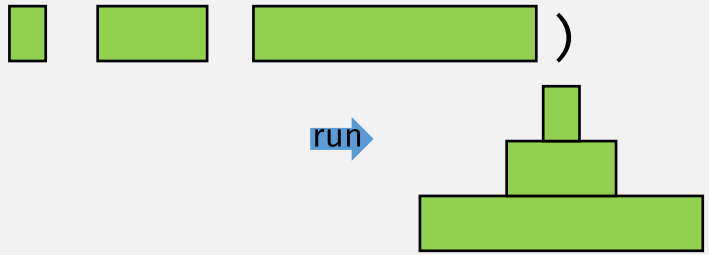
```
(+ 1 2 3 4)
```

- (fundamental) programming model: **arithmetic** expressions
  - But **not just numbers!**
  - When **"run", arithmetic expressions evaluate** to an **answer** or **value**

```
(string-append "hi" "world")
```

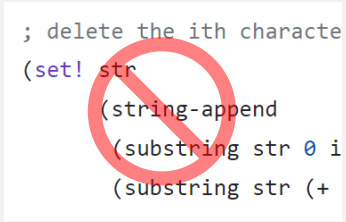
run "hi-world"

```
(above                    )
```

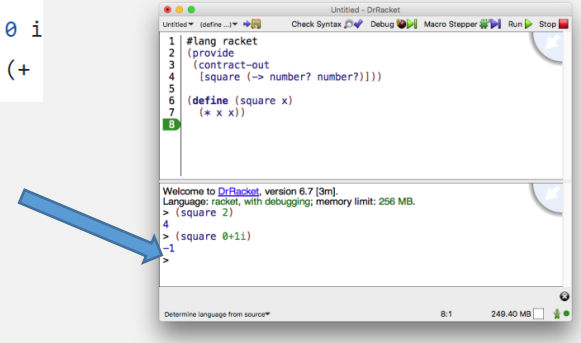


run

  - **No statements!**
    - E.g., "**assign**" or "**return**"



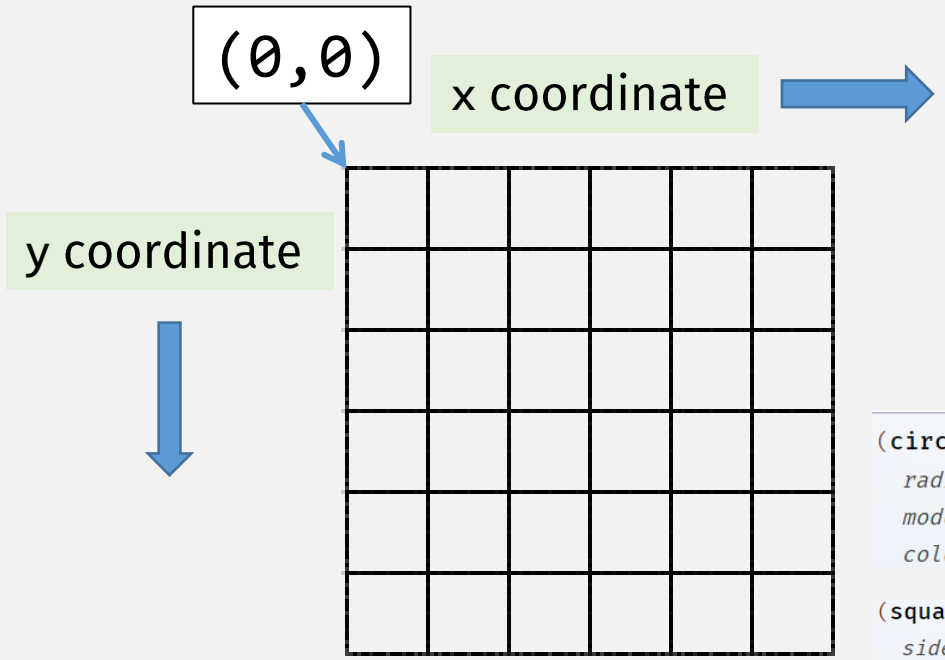
- Use the **REPL** ("interactions") for basic testing!



"image arithmetic"

Interlude: 2htdp/image library

`(require 2htdp/universe)`




```
(place-image image x y scene) → image?                    procedure
  image : image?
  x : real?
  y : real?
  scene : image?
```

Places *image* onto *scene* with its center at the coordinates (x,y) and crops the resulting image so that it has the same size as *scene*. The coordinates are relative to the top-left of *scene*.

```
(circle radius mode color) → image?
  radius : (and/c real? (not/c negative?))
  mode : mode?
  color : image-color?

(square side-len mode color) → image?
  side-len : (and/c real? (not/c negative?))
  mode : mode?
  color : image-color?
```

```
(place-image
 (circle 10 "solid" "red")
 0 0
 (square 40 "solid" "yellow"))
```

???

1 

2 

3 

4 

Functions

- `define` defines a function
  - The only non-expression you should use (for now)

```
(define (fn-name arg1 arg2)
    … body-expression …)
```

  - NOTE: `(define const-name expression)` defines a constant (different paren syntax)

- `lambda`
  - (anonymous) function expression
  - Function position in function call is just another expression!
  - `((lambda (x) (+ x 1)) 10)`
  - `((lambda (f) (f f)) (lambda (f) (f f)))` Warning!

- Predicates?
  - Function that **evaluates** to `true` or `false` when **called** or **applied**

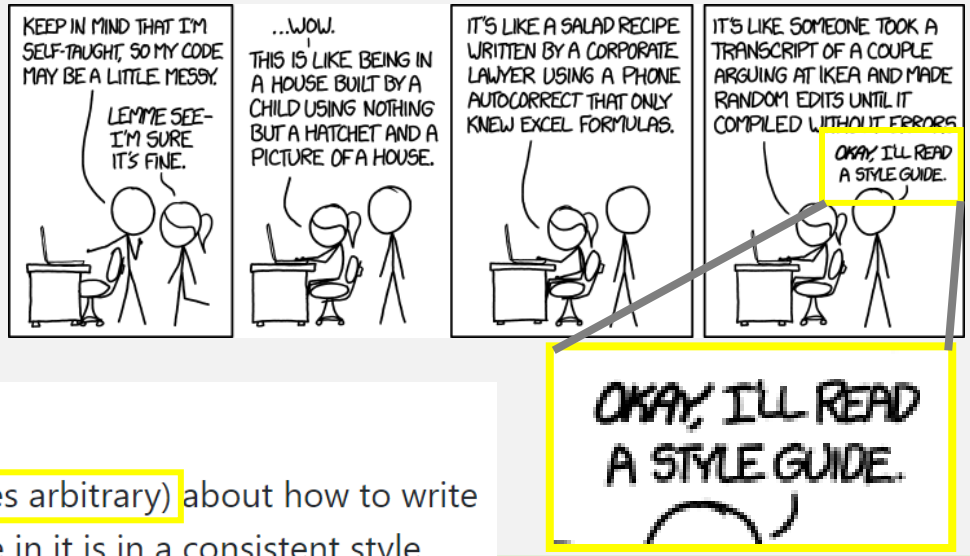
Programs

- **Programs** are a **sequence of** `defines` and **expressions**
  - One of them could be a "`main`" entry point

- When **program is "run", each is evaluated** to get an **answer** / **value**
  - similar to "reduction" in math

Style

- Critical for writing readable code, e.g.



Google Style Guides

Every major open-source project has its own style guide: a set of conventions (sometimes arbitrary) about how to write code for that project. It is much easier to understand a large codebase when all the code in it is in a consistent style.

"Style" covers a [illegible] bles" to "never use exceptions." This [illegible] If you are modifying a project that or[illegible] apply to that project.

- AngularJS S[illegible]
- Common Li[illegible]
- C++ Style Guide
- C# Style Guide
- Go Style Guide
- HTML/CSS Style Guide
- JavaScript Style Guide
- Java Style Guide
- Objective-C Style Guide
- Python Style Guide

Airbnb JavaScript Style Guide() {

A mostly reasonable approach to JavaScr[illegible]

Every company / organization you join will have "rule book" for writing code that you [illegible]

Microsoft | Learn Documentation Training Certifications Q&A Code Sampl

.NET Languages Features Workloads APIs Resources

Learn / .NET / C# guide / Fundamentals /

Common C# code conventions

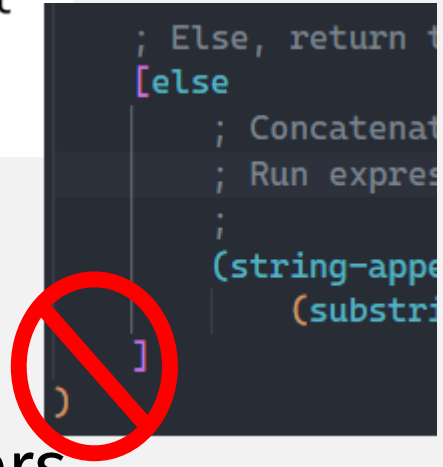
A code standard is essential for maintaining code readability, consistency, and collaboration within a development team. Following industry practices and established

Style: This Class

https://docs.racket-lang.org/style/index.html

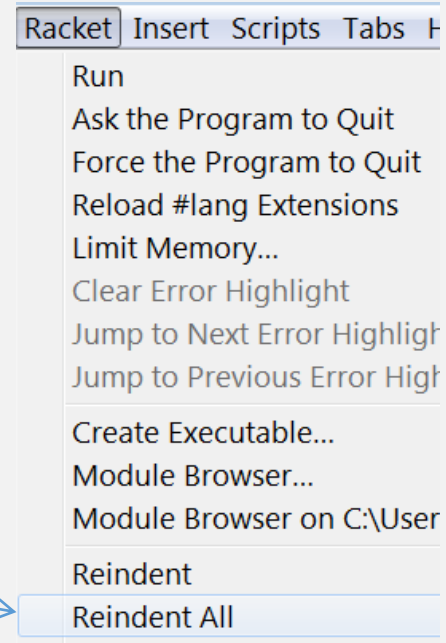
Racket

How to Program Racket: a Style Guide



A few tips

- Closing parens do not get their own line
- code width 80-100 columns
- Use dashes to separate multi-word identifiers (no underscore or CamelCase):
  - `string-append`
- Use DrRacket auto-indenter



Style: Git commits

- Git commits must also be **readable** (concise and informative)

How to Write a Git Commit Message

Commit messages matter. Here's how to write them well.

The seven rules of a great Git commit message

1. Separate subject from body with a blank line
2. Limit the subject line to 50 characters
3. Capitalize the subject line
4. Do not end the subject line with a period
5. Use the imperative mood in the subject line
6. Wrap the body at 72 characters
7. Use the body to explain *what* and *why* vs. *how*

A properly formed Git commit subject line s
complete the following sentence:

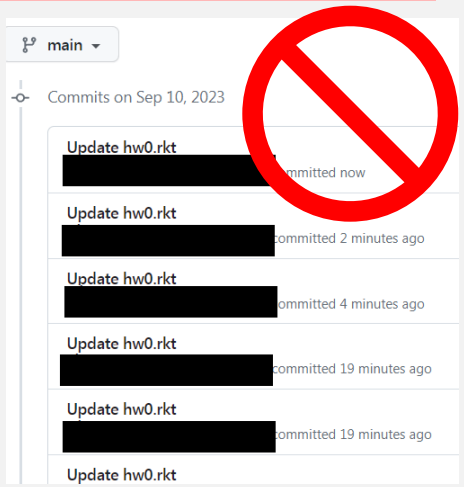
- If applied, this commit will *your subject line here*

For example:

- If applied, this commit will *refactor subsystem X for readability*
- If applied, this commit will *update getting started documentation*
- If applied, this commit will *remove deprecated methods*
- If applied, this commit will *release version 1.0.0*
- If applied, this commit will *merge pull request #123 from user/branch*

Notice how this doesn't work for the other non-imperative forms:

- If applied, this commit will ~~*fixed bug with Y*~~
- If applied, this commit will ~~*changing behavior of X*~~
- If applied, this commit will ~~*more fixes for broken stuff*~~
- If applied, this commit will ~~*sweet new API methods*~~



In-class exercise

- Using **`2htdp/image`** library:
  write a Racket expression that builds a "Traffic Light" image
- Put in file: **`in-class-02-03-<Lastname>-<Firstname>.rkt`**

Submit In-class work

- Join "**in-class" team** at:
  `https://github.com/orgs/cs450s26/teams/in-class`
- Commit file to this repo:
  `https://github.com/cs450s26/in-class-02-03`
- (May need to **`merge`** or **`pull + rebase`** if someone **push**es before you)