Development Environment

• I keep two windows open:
  • (a) an editor (Dr. Racket, emacs, VSCode, …)
  • (b) a command-line application (iTerm2)
Starting my development

• Every day when I begin my work I:
  • (a) open a new tab in the command line
  • (b) navigate to the project folder I want
  • Everything kept in git, so this is a git repo
Useful commands

• When I open up the command line, I’m in my **home** directory
• Use **cd** to change into the directory I want
• Use **tab completion** *always* when I use the shell
  • You should too!
Globs

• You can use search patterns ("globs") with most commands
• Regular-expression-like language (not standard)
• Lets me search *p1* to say “find anything with p1 in it”
Git status

• After getting into the work directory, I use “git status” to see what’s new

• Shows any uncommitted work
clear

• I hate seeing too much text on the screen
Running the tests

• Once in a while, I’ll run the tests
  • Always use **python3**
    • Old python is python 2, it is now dead
  • Run it from the command line
    • Same project folder that holds our git repo
• I encourage you to go **read** tester.py
• But it uses several helper scripts
Editing the code

• Several choices:
  • Emacs / vi in the terminal
  • Probably want side-by-side terms
Editing the code

- Most students will simply use Dr. Racket **and** a terminal
- This is fine—keep them both side-by-side
- (Switch between with command-tab on MacOS, …)
Test Always

• Whenever you do *something*, test it as **fast as possible**
• Otherwise you will lose context, context is crucial for bug finding!
• Get in the habit of pressing “run” a bunch
• Even if you run no tests, it does “rough check” of syntactic correctness
• Type tests in the REPL “manually” for small things, use the terminal to run larger tests
Other Editors

VSCode is worth trying
Derived Types

- **S-expressions** *(symbolic expression)*
  - Untyped lists that generalize neatly to trees:
    
    (this (is an) s expression)

- Computer represents these as **linked** structures
  
  - Cons cells of head & tail *(cons 1 2)*
Derived Types

• Racket also has **structural** types
  • Defined via `struct`; aids robustness
  • We will usually prefer agility of “tagged” S-expressions
• Also an elaborate object-orientation system (we won’t cover)
The function \texttt{cons} builds a cons cell

\[(\text{cons} \ 0 \ 1)\]
The function \texttt{car} gets the left element \((\texttt{car \ (cons 0 1)})\) is \(0\)

\[\text{\begin{tikzpicture}
  \node (0) at (0,0) {0};
  \node (1) at (1,0) {1};
  \draw[->] (0) -- (1);
\end{tikzpicture}}\]
The function \texttt{cdr} gets the left element

\[(\texttt{cdr} (\texttt{cons} \, 0 \, 1)) \text{ is } 1\]
At runtime, each cons cell sits at an address in memory

\[ \text{cdr (cons 0 1)} \] is 1

0x700000032acd1200
In fact, numbers are also stored in memory locations. They are thus said to be a "boxed" type.
Actually, every Racket variable stores a value in some “box” (i.e., memory location)

```
(define x 23)
(displayln x)
(set! x 24)
(displayln x)
```

```
0x700000033dea2280
```

```
X  23
```
Actually, every Racket variable stores a value in some "box" (i.e., memory location)

```
(define x 23)
(displayln x)
(set! x 24)
(displayln x)
```

0x700000033dea2280

```
x
```

23

Console output...
> 23
Actually, every Racket variable stores a value in some “box” (i.e., memory location)

```
(define x 23)
(displayln x)
(set! x 24)
(displayln x)
```

0x700000033dea2280

x 24

x’s value **changes** to 24
Vectors (similar to arrays) are mutable, and give \( O(1) \) indexing and updating.
Unless we say otherwise, you should avoid using set!, any use will be at your own risk

Similarly, avoid vector-set!, hash-set!, ...

Using set! will, in CIS352, lead to hard-to-debug code that will make it much harder for instructors to understand your code
Pairs enable us to build **linked lists** of data

\[
(\text{cons} \ 1 \ (\text{cons} \ 0 \ \textbf{‘}()))
\]

This is how Racket represents lists in memory
Note that in Racket, the following are equivalent

(cons 2 (cons 1 (cons 0 ’())))
’(2 1 0)

But the following is called an **improper list**

(cons 2 (cons 1 0))
’(2 1 . 0)

Dot indicates a cons cell of a left and right element
Also can build **compound** expressions

‘(this (is an) s expression)
Also can build **compound** expressions

`'(this (is an) s expression)`
"this"'s 'expression

"is"'an"
Draw the cons diagram for…
• (cons 0 (cons 3 4))
• Is this a list? If not, what is it?
• (cons 0 (cons 3 (cons 4 '())))
• Is this a list? If not, what is it?
(cons 0 (cons 3 4))

This is not a list (an improper list)
(cons 0 (cons 3 (cons 4 '())))