Functional Programming
What makes a PL functional?

✧ They provide abstractions over functions
What makes a PL functional?

- They provide abstractions over functions
- They treat functions like other values in the language
What makes a PL functional?

- They provide abstractions over functions
- They treat functions like other values in the language
- They emphasize recursion over iteration
What makes a PL functional?

- They provide abstractions over functions
- They treat functions like other values in the language
- They emphasize recursion over iteration
- They do not allow mutation
Mutation: overwriting the value of a variable or data structure.

State: a mechanism for keeping track of the current values associated with variables.

How is state different than memory?
Lambda Calculus

\[ \lambda y. \lambda x. y + x \]

Expresses a function that takes two arguments, \(x\) and \(y\), and adds them.

- Lambdas bind variables
- Lambda calculus describes computation using the concepts of function application, substitution, binding, and scope.
- There’s no mutation in lambda calculus.

Turing showed that the classes of functions defined by \(\lambda\)-calculus and Turing machines coincide.
Racket is Turing-complete (even without mutation)

We just have to learn to think functionally
Welcome to Racket

You are now a Racketeer....

xkcd
Dr. Racket

Welcome to DrRacket, version 6.12 [3m].
Language: racket, with debugging; memory limit: 128 MB.
# Basic Datatypes

<table>
<thead>
<tr>
<th>Booleans</th>
<th>Numbers</th>
</tr>
</thead>
<tbody>
<tr>
<td>#t</td>
<td>1</td>
</tr>
<tr>
<td>#f</td>
<td>1/2</td>
</tr>
<tr>
<td>Strings</td>
<td>1.0</td>
</tr>
<tr>
<td>“hi”</td>
<td></td>
</tr>
<tr>
<td>“h”</td>
<td></td>
</tr>
<tr>
<td>Characters</td>
<td></td>
</tr>
<tr>
<td># \h</td>
<td></td>
</tr>
<tr>
<td># \l</td>
<td></td>
</tr>
</tbody>
</table>
Lists

(list "apple" "banana" "carrot")

(list 1 2 3)

(list 1 "carrot" 3 #t "cucumber")
Lists

In Racket, lists are recursively defined: a list is either null, or a pair whose second item is a list.

Lists have two key methods: **first** and **rest**

```
> (first (list 1 2 3))
1

> (rest (list 1 2 3))
(list 1 2)
```
Control Flow

(if (= x 5) test
  #t value if true
  #f) value if false

(cond ((= x 0) (printf “x is 0”))
  ((= x 1) (printf “x is 1”))
  (else (printf “x is greater than 1”))))
Why are there so many parentheses?

A GOD’S LAMENT

SOME SAID THE WORLD SHOULD BE IN PERL;
SOME SAID IN LISP.
NOW, HAVING GIVEN BOTH A WHIRL,
I HELD WITH THOSE WHO FAVOURED PERL.
BUT I FEAR WE PASSED TO MEN
A DISAPPOINTING FOUNDING MYTH,
AND SHOULD WE WRITE IT ALL AGAIN,
I’D END IT WITH
A CLOSE-PAREN.
Syntax

✦ Leaf: a value that can’t be evaluated any further (also called an “atomic value” or a “literal”)
✦ Leaves are unparenthesized in Racket
✦ Every non-leaf node in the syntax tree is marked by a pair of parentheses
✦ **Special forms** have a keyword after the open parenthesis: (if e1 e2 e3)
✦ Most other parentheses mark function calls
Variable definitions

- Syntax: `(define id e)`
- Example:

  ```scheme
  (define x 1)
  > x
  1
  ```
Functions

- Syntax: `(define (id) e)`
- Examples:

  ```
  (define (add)
   (+ 10 10))
  ```
Functions

- Syntax: `(define (id) e)`
- Examples:

  `(define (add)
   (+ 10 10))`

  `(define (hello-world)
   (printf "Hello world!"))`
Functions

- Syntax: `(define (id) e)`
- Examples:

  `(define (add)
   (+ 10 10))`

  `(define (hello-world)
   (display "Hello world!"))`

⚠️ Warning: side effect ⚠️
Side effects

- **Side effect**: any observable effect other than producing a value
- Functional programming languages tend to avoid side effects (mutation is a kind of side effect)
- Side effects make it **harder to reason formally** about a program’s behavior
- However, printing is very useful!
Racket printing

- What’s the difference between `display`, `write`, and `print`?
- What does `displayln` do?
Documentation

✦ Racket Guide:
  - https://docs.racket-lang.org/guide/index.html

✦ Racket Reference:
  - https://docs.racket-lang.org/reference/index.html
Common Racket mistakes
Common Racket mistakes

1. Wrap leaf values in parens: (17)
2. Use operators in infix rather than prefix position
3. Put arguments in parentheses with function name outside
4. Use unexpected keywords
5. Omit parentheses for non-leaf node