Racket

- **Dynamically typed**: variables are untyped, values typed
- **Functional**: Racket emphasizes functional style
  - Compositional—emphasizes black-box components
  - Immutability—requires automatic memory management
- **Imperative**: Racket allows data to be modified, in carefully considered cases, but doesn’t emphasize “impure” code
- **Object-oriented**: racket has a powerful object system
- **Language-oriented**: Racket is really a language toolkit
- **Homoiconic**: Code is data; the primary data structure of Scheme, and LISP-family languages, is the *linked list*, written as *s-expressions*, & Scheme code is explicitly written as lists.
Calculating the slope of a line in Racket

(define (calculate-slope x0 y0 x1 y1)
  (/ (- y1 y0) (- x1 x0)))
(define (calculate-slope x0 y0 x1 y1)
  (/ (- y1 y0) (- x1 x0)))

Prefix notation
Functions defined via prefix notation, too

(define (calculate-slope x0 y0 x1 y1)
  (/ (- y1 y0) (- x1 x0)))
(define (calculate-slope x0 y0 x1 y1) (/ (- y1 y0) (- x1 x0)))

(calculate-slope 0 0 3 2)

Calls to user-defined functions also in prefix notation
(define (calculate-slope x0 y0 x1 y1)
  (/ (- y1 y0) (- x1 x0)))

(calculate-slope 0 0 3 2)

Note: preferred style puts closing parens at end of blocks
Basic Types

• **Numeric tower.** Numeric types gracefully degrade
  • E.g., (* (/ 8 3) 2+1i) is 16/3+8/3i
  • Note that 2+1i is a **literal** value, as is 2.3

• **Strings** and **characters** ("foo" and \\a)

• **Booleans** (#t and #f) including logical operator (e.g., or)
  • Note that operators “short circuit”

• **Symbols** are interned strings ‘foo
  • Implicitly only one copy of each, unlike (say) strings

• The #<void> value (produced by (void))
Compute the sum of the following:

• $2/3$ and $1.5$
• $3+8i$ and $3i$
• $0$ and positive infinity ($+\infty.0$)
Compute the sum of the following:

- $(+ \frac{2}{3} 1.5)\quad 2.16666666666666665$ (N.B., result is **inexact**)
- $(+ 3+8i 3i)\quad 3+11i$
- $(+ 0 +\text{inf}.0)\quad +\text{inf}.0$
Forms

• A **form** is a recognized syntax in the language
  • (if ...), (and ...) are forms, but +, list refer to functions
  • You can define new forms too! More on this later...
• Scheme prefers to give a small number of general forms.
• The tag just after the open-paren determines the form:
  • (define foo value) — Define a variable
  • (define (foo a0 a1 ...) body) — Define a function
  • (if guard e-true e-false), (or e0 e1 ...), etc
• Otherwise, by default, each pair of parens is a **call site**.
Define a function that takes an argument, x, and returns:

• x times 2, if x is less than 0
• x times -2 otherwise

**Hint:** use\((< x y)\) for comparison
(define (f x)
  (if (< x 0)
   (* 2 x)
   (* -2 x))))
Define a function that takes an argument, $x$, and returns:

- $x$ divided by 2, if $x$ is even
- $x$ times 3 plus 1, if $x$ is odd

**Hint:** use $=$ and modulo to check if $x$ is even/odd
(define (collatz x)
  (if (= 0 (modulo x 2))
    (/ x 2)
    (+ 1 (* 3 x)))))
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 (pairs) of a head and a tail (cons 1 2)
• Racket also has **structural** types (defined via structs)
  • Defined via struct; aids robustness
  • We will usually prefer agility of “tagged” S-expressions
• Also an elaborate object-orientation system (we won’t cover)
Example

The function *cons* builds a cons cell

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

\[
\begin{array}{c}
\text{0} \\
\downarrow \\
\text{1}
\end{array}
\]

\[
\left( \lambda(x) \, (x \, x) \right) \\
\left( \lambda(x) \, (x \, x) \right)
\]
Example

\((\text{car} \ (\text{cons} \ 0 \ 1))\) is 0

The function \textbf{car} gets the left element
(cdr (cons 0 1)) is 1

The function cdr gets the right element
At runtime, each cons cell sits at an address in memory

```
(cons 0 1)
0x700000032acd1200
0
1
```
In fact, numbers are also stored in memory locations. They are thus said to be a “boxed” type
(define x 23)
(displayln x)
(set! x 24)
(displayln x)

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

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

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

```
0x700000033dea2280
```

Changes x’s value to 24
Actually, every Racket variable stores a value in some "box" (i.e., memory location)

Now prints 24
Vectors (similar to arrays) are mutable, and give $O(1)$ indexing and updating.
In this class, you will not be allowed to use \texttt{set!} or \texttt{vector-set!} unless explicitly noted.

Code that uses \texttt{set!} may be denied full credit.
Pairs enable us to build **linked lists** of data

\[(\text{cons } 1 \ (\text{cons } 0 \ '())\)]

This is how Racket represents lists in memory.
Example

Note that in Racket, the following are equivalent

\[
(\text{cons } 2 \ (\text{cons } 1 \ (\text{cons } 0 \ '()))
\]

\['(2 \ 1 \ 0)\]

But the following is called an \textbf{improper list}

\[
(\text{cons } 2 \ (\text{cons } 1 \ 0))
\]

\['(2 \ 1 \ . \ 0)\]

Dot indicates a cons cell of a left and right element
Example

Also can build **compound** expressions

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

`(this (is an) s expression)`
Example

Empty list


diagram showing the structure of the expression

```
((\x (x x)) (\x (x x)))
```

This expression is an empty list.
Example

Note link to compound subexpression

\[
\left( \lambda (x) (x \ x) \right) \\
\left( \lambda (x) (x \ x) \right)
\]

‘this

‘is

‘an

‘s ‘expression

‘()
Exercise

Draw the cons diagram for the following...
• (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?
Exercise

Draw the cons diagram for the following...

• (cons 0 (cons 3 4)) — Drawn on board
• Is this a list? If not, what is it?
  • No, not a list, but an improper list, no empty list at end
• (cons 0 (cons 3 (cons 4 ‘()))) — Drawn on board
• Is this a list? If not, what is it?
  • Yes, this is a list
• Identifiers refer to their most proximate syntactic binding
  • I.e., Racket is **statically scoped**; more later…

• Can create local bindings with the **let** form:
  • `(let ([x 0] [y 1]) body)`  
    x is bound to 0, y to 1, in body
  • Note that y cannot reference x! Otherwise you want “sequential let”, the **let*** form
  • `(let ([x 23] [y (* 2 x)] (+ y 2))`  
    undefined variable x!
What is the value of the following expression?

$$(\text{let } ([a \ 1] \\
[\text{b} \ 2]) \\
(\text{let } ([\text{b} \ 3] \\
[\text{c} \ 4]) \\
(+ \ a \ b \ c)))$$
What is the value of the following expression?

(let ([a 1] [b 2]))
  (let ([b 3] [c 4])
    (+ a b c)))

8

The second definition of b **shadows** the first b. At the point where + is invoked on three values, b is bound most proximately to 3.
What is the value of the following expression?

(let ([a 1]
      [b 2])
  (let ([b 3]
        [c (+ a b)])
    c))
What is the value of the following expression?

(let ([a 1]
      [b 2])
  (let ([b 3]
         [c (+ a b)])
    c))

Although the second definition of b *shadows* the first b, when defining c, the value of b is still 2!

The new binding only takes effect in the body of the let form.
Use `let*` to evaluate the following mathematical expression (without simplifying it), where $x$ is 4:

$$\left((x \times 2) + (x \times 2) + (x \times 2)\right)^2$$
Use `let*` to evaluate the following mathematical expression (without simplifying it), where $x$ is 4:

$$\left( (x \times 2) + (x \times 2) + (x \times 2) \right)^2$$

```scheme
(let* ([x 4]
        [y (* x 2)]
        [z (+ y y y)])
  (* z z))
```
What does the following code compute?

```
(define (foo x) 1)

(let* ([f foo]
       [f (f 2)])
  (* f (let ([f 3])
         (+ f (foo f)))))
```
What does the following code compute?

```
(define (foo x) 1)

(let* ([f foo]
        [f (f 2)])
  (* f (let ([f 3])
        (+ f (foo f)))))
```
Exercise

For each variable use within the following code, identify the variable’s proximate binder

```
(define (foo x) 1)

(let* ([f foo]
        [f (f 2)])
  (* f (let ([f 3])
       (+ f (foo f)))))
```
Exercise

For each variable use within the following code, identify the variable’s proximate binder

\[
\text{(define (foo } x) 1) \\
\text{(let* ( [f foo] [f (f 2)])} \\
(* f (let ([f 3]) (+ f (foo f)))))
\]