Racket: Functions & Lists (& Recursion)
What kind of data structure is this?
Racket: operations (s-expressions)

\((op \ arg_1 \ arg_2 \ldots \ arg_n)\)

• **Rules:**
  • the operation always comes first
  • its arguments (if there are any) follow the operation
  • no commas between arguments
  • everything goes between parentheses

• **Common mistakes:**
  • forgetting parentheses
  • rational vs. integer division (/ vs. quotient)
  • equality (= vs. equal?)
Racket: “variables”

They’re called variables, but we won’t vary them (i.e., their values are constant).

```racket
(let* ( [var₁ expr₁] 
       ... 
       [varₙ exprₙ])
   body)
```

“bind” a value to a variable

“scope” of variables
Welcome to DrRacket, version 6.6 [3m].
Language: racket, with debugging; memory limit: 128 MB.

> (let* ([x 30]
       [y 12])
    (+ x y))
42
> x
x: undefined;
cannot reference an identifier before its definition
> (let* ([x 30]
       [z 12])
    (+ x y))

y: undefined;
cannot reference an identifier before its definition
>
idiom: if you have more than one condition, use cond

```
(if conditional-expr
  true-expr
  false-expr)
```

```
(cond  
  [condition\_1 expr\_1]  
  ...  
  [condition\_n expr\_n]  
  [else else-expr])
```

this is the most common form of cond
Welcome to DrRacket, version 6.6 [3m].
Language: racket, with debugging; memory limit: 128 MB.

> (if (= 1 1)
   "math works"
   "math is broken")
"math works"

> (cond [(> 42 0) "positive"
          [(< 42 0) "negative"
           [else "zero"])
"positive"

>
Racket: functions

\[
\text{define } (function-name \ parameter_1 \ldots \ parameter_n) \ body
\]
#lang racket

(define (california temp)
  (cond [(> temp 75) "hot!"]
        [(< temp 65) "cold!"]
        [else "ok"]))

Welcome to DrRacket, version 6.2.1 [3m].
Language: racket [custom]; memory limit: 128 MB.
> (california 10)
"cold!"
> (california 70)
"ok"
> (california 79)
"hot!"
Write tests first!

Using rackunit

```racket
#lang racket

;;; this line gives us access to the testing library

;;; int-average

;;; computes the average of two numbers, using integer division
;;; inputs: x & y, two integers
;;; outputs: the integer average of the two inputs
(define (int-average x y) 0)

; tests
(check-equal? (int-average 0 0) 0)
(check-equal? (int-average 0 2) 1)
(check-equal? (int-average 4 6) 5)
(check-equal? (int-average 1 1) 1)
```

---

**FAILURE**

| actual: | 0 |
| expected: | 1 |
| name: | check-equal? |
| location: | (#<path:/Users/ben/Documents/work/teaching/courses/CS42/fall 2016/class/05_1 – Racket Intro/code/average.rkt> 19 0 497 34) |
| expression: | (check-equal? (int-average 1 1) 1) |

---

**FAILURE**

| actual: | 0 |
| expected: | 1 |

---

*Check failure*
Write tests first!

Using rackunit

```racket
#lang racket

;; int-average: computes the average of two numbers, using integer division
;; inputs: x & y, two integers
;; outputs: the integer average of the two inputs
(define (int-average x y)
  (quotient (+ x y) 2))

;; tests
(check-equal? (int-average 0 0) 0)
(check-equal? (int-average 0 2) 1)
(check-equal? (int-average 4 6) 5)
(check-equal? (int-average 1 1) 1)
(check-equal? (int-average 1 2) 1)
```
Writing functions: let’s practice

Write a function that computes the average of two numbers.

```racket
(require rackunit)

;; comments omitted, to fit on slide
;; (but we should always document our functions)
(define (int-average x y)
  (quotient (+ x y) 2))

; tests
(check-equal? (int-average 0 0) 0)
(check-equal? (int-average 0 2) 1)
(check-equal? (int-average 4 6) 5)
(check-equal? (int-average 1 1) 1)
(check-equal? (int-average 1 2) 1)
```
Racket lists
## Creating lists in Racket

<table>
<thead>
<tr>
<th>Syntax</th>
<th>Printed representation</th>
<th>Semantics</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>empty</strong></td>
<td><code>'(())</code></td>
<td></td>
</tr>
<tr>
<td><strong>(list &lt;value1&gt; ... &lt;valueN&gt;)</strong></td>
<td><code>'(2 3)</code></td>
<td></td>
</tr>
<tr>
<td><strong>(cons &lt;value&gt; &lt;list&gt;)</strong></td>
<td><code>'(1 2 3)</code></td>
<td></td>
</tr>
</tbody>
</table>

- **Syntax**: what we write
- **Printed representation**: what Racket prints
- **Semantics**: what it means

1. **empty**: make an empty list
   - Syntax: `'(())`
   - Printed representation: `'(())`
   - Semantics: empty list

2. **(list <value1> ... <valueN>)**: make a list with N values
   - Syntax: `'(2 3)`
   - Printed representation: `'(2 3)`
   - Semantics: list `[2, 3]`

3. **(cons <value> <list>)**: add an element to the front of a list
   - Syntax: `'(1 2 3)`
   - Printed representation: `'(1 2 3)`
   - Semantics: list `[1, 2, 3]`
Creating lists: let’s practice
write down the answers as either a drawing or a Racket expression

1. \(\text{list} \ 3 \ 2 \ 1\) \(\leftarrow\) draw the picture

2. \(\text{cons} \ 3 \ \text{(list} \ 2 \ 1\)) \(\leftarrow\) draw the picture

3. \(\text{draw the expression}\)

4. \'(1) \(\leftarrow\) write the expression that makes Racket display this

5. \(\text{list} \ 3 \ \text{(list} \ 2 \ 1\) \(\text{0}\)) \(\leftarrow\) draw the picture
Aside: we don’t actually need list!

list is “syntactic sugar” for one or more calls to cons

(list 1 2 3)

is the same as

(cons 1 (cons 2 (cons 3 empty)))
Accessing Racket lists

```
def first(x):
    return x[0]

def rest(x):
    return x[1:]
```
Accessing lists: let’s practice

Assume the variable L has the value ' (1 2 3). Fill in the table.

<table>
<thead>
<tr>
<th>result</th>
<th>expression that uses L to compute result</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>(first L)</td>
</tr>
<tr>
<td>(2 3)</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td></td>
</tr>
<tr>
<td>(3)</td>
<td></td>
</tr>
</tbody>
</table>
### Accessing lists: let’s practice

Assume the variable L has the value `(1 2 3)`. Fill in the table.

<table>
<thead>
<tr>
<th>result</th>
<th>expression that uses L to compute result</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td><code>(first L)</code></td>
</tr>
<tr>
<td><code>(2 3)</code></td>
<td><code>(rest L)</code></td>
</tr>
<tr>
<td>2</td>
<td><code>(first (rest L))</code></td>
</tr>
<tr>
<td><code>(3)</code></td>
<td><code>(rest (rest L))</code></td>
</tr>
</tbody>
</table>
Watch out!

Don’t do these things (and if you accidentally do, know how to recognize them)

> (cons 1 2)
'(1 . 2)  ➔ not a list!

This expression builds a *pair*.

A pair is **not** a list.

You can’t call *first* on it.
You can’t call *rest* on it.

> '(list 1 2)
'(list 1 2)

This expression builds a list whose first element is 'list!

For more info, see: docs.racket-lang.org/guide/Pairs__Lists__and_Racket_Syntax.html
Take-home midterm #1

Available: now

Must return by: next Sunday (10/2) at 5pm

Time-limit: 75 minutes

Covers: everything up to and including this week automata, circuits, and assembly

Resources: one, 8½ x 11 sheet of notes (double-sided)

Honor code: don’t discuss exam questions

There will be a small assignment this week, on Racket.