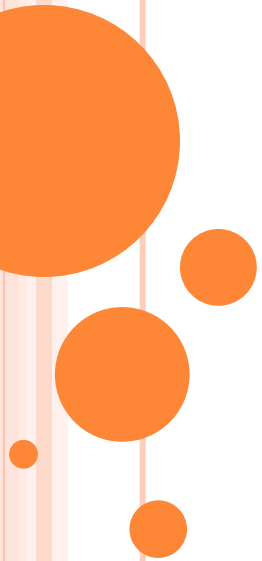

Racket 2

CS A250 – C++ Programming Language 2

Arithmetic Operations

- Racket uses **prefix notation** for arithmetic operations
  - Also called **Polish notation**
- Basic arithmetic operations are +, -, *, and /
- There is **no** mod (%)
- **No** distinction between **whole numbers** and **decimals**.

Arithmetic Operations (cont.)

- Racket uses **prefix notation**, where operators
  - Are written ***before*** their operands
  - Act on the two **nearest values** on the **right**

```
> (+ 1.2 3)
4.2

> (- 5 1)
4

> (* 2 3)
6

> (/ 6 2)
3
```

Prefix Notation

- **Prefix notation** places **operators** to the **left** of their operands

1 + 2 is equivalent to + 1 2

Conversion: Infix to Prefix

- How to convert an expression from **infix to prefix?**

```
15 / (7 – (1 + 2)) • 3 – (2 + 1 + 4)
```

Conversion: Infix to Prefix

- How to convert an expression from **infix to prefix?**

```
15 / (7 – (1 + 2)) • 3 – (2 + 1 + 4)
```

- Add parenthesis according to the order of precedence to specify the order of operations. If two operators have same precedence, simply add parenthesis to enclose only one operator

```
15 / (7 – (1 + 2)) • 3 – (2 + 1 + 4)
((15 / (7 – (1 + 2))) • 3) – (2 + (1 + 4))
```

Conversion: Infix to Prefix

- How to convert an expression from **infix to prefix?**

```
15 / (7 – (1 + 2)) • 3 – (2 + 1 + 4)
```

- Move each operator to the corresponding **left**

```
((15 / (7 – (1 + 2))) • 3) – (2 + (1 + 4))
((15 / (7 – (+ 1 2))) • 3) – (2 + (1 + 4))
```

Conversion: Infix to Prefix

- How to convert an expression from **infix to prefix?**

```
15 / (7 – (1 + 2)) • 3 – (2 + 1 + 4)
```

- Move each operator to the corresponding **left**

```
((15 / (7 – (1 + 2))) • 3) – (2 + (1 + 4))
((15 / (7 – (+ 1 2))) • 3) – (2 + (1 + 4))
((15 / (– 7 (+ 1 2))) • 3) – (2 + (1 + 4))
```

Conversion: Infix to Prefix

- How to convert an expression from **infix to prefix?**

```
15 / (7 – (1 + 2)) • 3 – (2 + 1 + 4)
```

- Move each operator to the corresponding **left**

```
((15 / (7 – (1 + 2))) • 3) – (2 + (1 + 4))
((15 / (7 – (+ 1 2))) • 3) – (2 + (1 + 4))
((15 / (– 7 (+ 1 2))) • 3) – (2 + (1 + 4))
((/ 15 (– 7 (+ 1 2))) • 3) – (2 + (1 + 4))
```

Conversion: Infix to Prefix

- How to convert an expression from **infix to prefix?**

```
15 / (7 – (1 + 2)) • 3 – (2 + 1 + 4)
```

- Move each operator to the corresponding **left**

```
((15 / (7 – (1 + 2))) • 3) – (2 + (1 + 4))
((15 / (7 – (+ 1 2))) • 3) – (2 + (1 + 4))
((15 / (– 7 (+ 1 2))) • 3) – (2 + (1 + 4))
((/ 15 (– 7 (+ 1 2))) • 3) – (2 + (1 + 4))
((/ 15 (– 7 (+ 1 2))) • 3) – (2 + (1 + 4))
( • (/ 15 (– 7 (+ 1 2))) 3) – (2 + (1 + 4))
```

Conversion: Infix to Prefix

- How to convert an expression from **infix to prefix?**

```
15 / (7 – (1 + 2)) • 3 – (2 + 1 + 4)
```

- Move each operator to the corresponding **left**

```
((15 / (7 – (1 + 2))) • 3) – (2 + (1 + 4))
((15 / (7 – (+ 1 2))) • 3) – (2 + (1 + 4))
((15 / (– 7 (+ 1 2))) • 3) – (2 + (1 + 4))
((/ 15 (– 7 (+ 1 2))) • 3) – (2 + (1 + 4))
((/ 15 (– 7 (+ 1 2))) • 3) – (2 + (1 + 4))
( • (/ 15 (– 7 (+ 1 2))) 3) – (2 + (1 + 4))
( • (/ 15 (– 7 (+ 1 2))) 3) – (2 + (+ 1 4))
```

Conversion: Infix to Prefix

- How to convert an expression from **infix to prefix?**

```
15 / (7 – (1 + 2)) • 3 – (2 + 1 + 4)
```

- Move each operator to the corresponding **left**

```
((15 / (7 – (1 + 2))) • 3) – (2 + (1 + 4))
((15 / (7 – (+ 1 2))) • 3) – (2 + (1 + 4))
((15 / (– 7 (+ 1 2))) • 3) – (2 + (1 + 4))
((/ 15 (– 7 (+ 1 2))) • 3) – (2 + (1 + 4))
((/ 15 (– 7 (+ 1 2))) • 3) – (2 + (1 + 4))
( • (/ 15 (– 7 (+ 1 2))) 3) – (2 + (1 + 4))
( • (/ 15 (– 7 (+ 1 2))) 3) – (2 + (+ 1 4))
( • (/ 15 (– 7 (+ 1 2))) 3) – (+ 2 (+ 1 4))
```

Conversion: Infix to Prefix

- How to convert an expression from **infix to prefix?**

```
15 / (7 – (1 + 2)) • 3 – (2 + 1 + 4)
```

- Move each operator to the corresponding **left**

```
((15 / (7 – (1 + 2))) • 3) – (2 + (1 + 4))
((15 / (7 – (+ 1 2))) • 3) – (2 + (1 + 4))
((15 / (– 7 (+ 1 2))) • 3) – (2 + (1 + 4))
((/ 15 (– 7 (+ 1 2))) • 3) – (2 + (1 + 4))
((/ 15 (– 7 (+ 1 2))) • 3) – (2 + (1 + 4))
( • (/ 15 (– 7 (+ 1 2))) 3) – (2 + (1 + 4))
( • (/ 15 (– 7 (+ 1 2))) 3) – (2 + (+ 1 4))
( • (/ 15 (– 7 (+ 1 2))) 3) – (+ 2 (+ 1 4))
– ( • (/ 15 (– 7 (+ 1 2))) 3) (+ 2 (+ 1 4))
```

Conversion: Infix to Prefix

- How to convert an expression from **infix to prefix?**

```
15 / (7 – (1 + 2)) • 3 – (2 + 1 + 4)
```

- Remove all parenthesis

```
– ( • (/ 15 (– 7 (+ 1 2))) 3) (+ 2 (+ 1 4))
```

```
– • / 15 – 7 + 1 2 3 + 2 + 1 4
```

Conversion: Prefix to Infix

- Now we will do the opposite conversion

```
- • / 15 - 7 + 1 2 3 + 2 + 1 4
```

From **prefix** to **infix**.

Conversion: Prefix to Infix

- Now we will do the opposite conversion

```
- • / 15 - 7 + 1 2 3 + 2 + 1 4
```

- Combine, using parenthesis, the pattern

```
operator-operand-operand
```

Conversion: Prefix to Infix

- Now we will do the opposite conversion

```
- • / 15 - 7 + 1 2 3 + 2 + 1 4
```

- Combine, using parenthesis, the pattern

```
operator-operand-operand
```

```
- • / 15 - 7 + 1 2 3 + 2 + 1 4
- • / 15 - 7 (+ 1 2) 3 + 2 (+ 1 4)
```

Conversion: Prefix to Infix

- Now we will do the opposite conversion

```
- • / 15 - 7 + 1 2 3 + 2 + 1 4
```

- Combine, using parenthesis, the pattern

```
operator-operand-operand
```

```
- • / 15 - 7 + 1 2 3 + 2 + 1 4
- • / 15 - 7 (+ 1 2) 3 + 2 (+ 1 4)
- • / 15 (- 7 (+ 1 2)) 3 (+ 2 (+ 1 4))
```

Conversion: Prefix to Infix

- Now we will do the opposite conversion

```
- • / 15 - 7 + 1 2 3 + 2 + 1 4
```

- Combine, using parenthesis, the pattern

```
operator-operand-operand
```

```
- • / 15 - 7 + 1 2 3 + 2 + 1 4
- • / 15 - 7 (+ 1 2) 3 + 2 (+ 1 4)
- • / 15 (- 7 (+ 1 2)) 3 (+ 2 (+ 1 4))
- • (/ 15 (- 7 (+ 1 2))) 3 (+ 2 (+ 1 4))
```

Conversion: Prefix to Infix

- Now we will do the opposite conversion

```
- • / 15 - 7 + 1 2 3 + 2 + 1 4
```

- Combine, using parenthesis, the pattern

```
operator-operand-operand
```

```
- • / 15 - 7 + 1 2 3 + 2 + 1 4
- • / 15 - 7 (+ 1 2) 3 + 2 (+ 1 4)
- • / 15 (- 7 (+ 1 2)) 3 (+ 2 (+ 1 4))
- • (/ 15 (- 7 (+ 1 2))) 3 (+ 2 (+ 1 4))
- (• (/ 15 (- 7 (+ 1 2))) 3) (+ 2 (+ 1 4))
```

Conversion: Prefix to Infix

- Now we will do the opposite conversion

```
- • / 15 - 7 + 1 2 3 + 2 + 1 4
```

- Combine, using parenthesis, the pattern

```
operator-operand-operand
```

```
- • / 15 - 7 + 1 2 3 + 2 + 1 4
- • / 15 - 7 (+ 1 2) 3 + 2 (+ 1 4)
- • / 15 (- 7 (+ 1 2)) 3 (+ 2 (+ 1 4))
- • (/ 15 (- 7 (+ 1 2))) 3 (+ 2 (+ 1 4))
- (• (/ 15 (- 7 (+ 1 2))) 3) (+ 2 (+ 1 4))
```

Conversion: Prefix to Infix

- Now we will do the opposite conversion

```
- • / 15 - 7 + 1 2 3 + 2 + 1 4
```

- Move all operators to the right of the first operand

```
- (• (/ 15 (- 7 (+ 1 2))) 3) (+ 2 (+ 1 4))

((15 / (7 - (1 + 2))) • 3) - (2 + (1 + 4))
```

Conversion: Prefix to Infix

- Now we will do the opposite conversion

```
- • / 15 - 7 + 1 2 3 + 2 + 1 4
```

- You may remove some parenthesis, but make sure you pay attention to the order of precedence.

```
((15 / (7 - (1 + 2))) • 3) - (2 + (1 + 4))
```

```
15 / (7 - (1 + 2)) • 3 - 2 + 1 + 4
```

This is our original expression.

Prefix Notation Advantages

- Other than being the standard notation for **Lisp** languages, **prefix notation** has the following advantages:
  - No need for parenthesis
  - Precedence rule is no longer relevant

Comparison Operators

- Prefix notation also applies to comparison operators

```
> (< 3 5)
#t

> (> 3 5)
#f

> (<= 2 3)
#t

> (>= 2 3)
#f

> (>= 2 2)
#t
```

Comparison Operators

- Prefix notation also applies to comparison operators

```
> (< 3 5)
#t

> (> 3 5)
#f

> (<= 2 3)
#t

> (>= 2 3)
#f

> (>= 2 2)
#t
```

Operators == and != are **NOT** defined in DrRacket.

Functions

FUNCTIONS

- Use **lambda** (**λ**) notation to implement **functions**
- Example:

```
> (define name of function
    (lambda (parameters)
       (expression)))
1
```

```
> (define my-first
    (lambda (lis)
       (first lis)))

> (my-first '(1 2 3))
1
```

"lis" is the parameter identifier

Function Naming

- **Function names**
  - Use **lower case** and separate words with a **dash** (-)

    remove-odds

    sum-even-elements
  - Can consist of letters, digits, and special characters, **except** parentheses
- **Racket** is **case-sensitive**

- Some implementations of **Lisp sub-languages**
  - Are case **in**sensitive
  - *Cannot* have function names that begin with a digit

Example 1

- Define a function ***my-second*** that returns the **second element** in a **list**
  - Where do you start? Think of the **function call**

```
> (my-second '(1 2 3))
```

Function call

```
> (my-second '(1 2 3))
```

Parameter is a list

Function name

Example 1 (cont.)

- Define a function ***my-second*** that returns the **second element** in a **list**
  - Which **expression** would we need to get the second?

```
> (my-second '(1 2 3))
```

Function call

Example 1 (cont.)

- Define a function ***my-second*** that returns the **second element** in a **list**
  - Which **expression** would we need to get the second?

```
> (my-second '(1 2 3))
```

Function call

```
> (define x '(1 2 3))
> (first (rest x))
2
```

But this expression would work only for **x**

Example 1 (cont.)

- Define a function ***my-second*** that returns the **second element** in a **list**
  - Let's look at the **function skeleton**

```
> (my-second '(1 2 3))
```

Function call

```
> (define x '(1 2 3))
> (first (rest x))
```

```
> (define name of function
    (lambda (parameters)
       (expression)))
```

Example 1 (cont.)

- Define a function ***my-second*** that returns the **second element** in a **list**
  - Substitute **function name** and **expression**

```
> (my-second '(1 2 3))
```

Function call

```
> (define x '(1 2 3))
> (first (rest x))
```

```
> (define my-second
    (lambda (lis)
      (first (rest lis))))
```

lis – Parameter name

Example 2

- Define a function ***area-rectangle*** that calculates the area of a rectangle
  - Think of the **function call**

```
> (area-rectangle '(2 3))
```

Function call
*(note that width and length are represented as a **list**)*

Example 2 (cont.)

- Define a function ***area-rectangle*** that calculates the area of a rectangle
  - Which **expression** would we need to find the area?

```
> (area-rectangle '(2 3))
```

Function call

Example 2 (cont.)

- Define a function ***area-rectangle*** that calculates the area of a rectangle
  - Which **expression** would we need to find the area?

```
> (area-rectangle '(2 3))
```

Function call

```
> (define d '(2 3))
> (* (first d) (first (rest d)))
6
```

Example 2 (cont.)

- Define a function ***area-rectangle*** that calculates the area of a rectangle
  - Let's look at the **function skeleton**

```
> (area-rectangle '(2 3))
```

Function call

```
> (define d '(2 3))
> (* (first d) (first (rest d))
```

```
> (define name of function
    (lambda (parameters)
       (expression)))
```

Example 2 (cont.)

- Define a function ***area-rectangle*** that calculates the area of a rectangle
  - Substitute **function name** and **expression**

```
> (area-rectangle '(2 3))
```

Function call

```
> (define d '(2 3))
> (* (first d) (first (rest d))
```

```
> (define area-rectangle
    (lambda (lis)
       (* (first lis) (first (rest lis))
```

lis – Parameter name

Example 2 (cont.)

- Define a function ***area-rectangle*** that calculates the area of a rectangle
  - Use (;) to add comments and then test your function

```
> ; sides will be in a list
> (define area-rectangle
    (lambda (lis)
      (* (first lis) (first (rest lis)))))

> (define dimensions '(4 5)

> (area-rectangle dimensions)
20
```

Example 2 (cont.)

- Define a function ***area-rectangle*** that calculates the area of a rectangle
  - We could also use *two* separate parameters

```
> ; using two separate parameters
> (define area-rectangle
    (lambda (x y)
      (* x y)))

> (area-rectangle 4 5)
20
```

2 parameters

Example 3 (cont.)

- Mixed expressions

```
> (define a 'Bob)
> (define b 'Jane)
> (define c '(Jane))
> (define d '(Bob Jane))

> (equal? a (my-second d))
```

Example 3 (cont.)

- Mixed expressions

```
> (define a 'Bob)
> (define b 'Jane)
> (define c '(Jane))
> (define d '(Bob Jane))

> (equal? a (my-second d))
#f
```

Example 3 (cont.)

- Mixed expressions

```
> (define a 'Bob)
> (define b 'Jane)
> (define c '(Jane))
> (define d '(Bob Jane))

> (equal? a (my-second d))
#f

> (equal? b (my-second d))
```

Example 3 (cont.)

- Mixed expressions

```
> (define a 'Bob)
> (define b 'Jane)
> (define c '(Jane))
> (define d '(Bob Jane))

> (equal? a (my-second d))
#f

> (equal? b (my-second d))
#t
```

Example 3 (cont.)

- Mixed expressions

```
> (define a 'Bob)
> (define b 'Jane)
> (define c '(Jane))
> (define d '(Bob Jane))

> (equal? a (my-second d))
#f

> (equal? b (my-second d))
#t

> (equal? c (my-second d))
```

Example 3 (cont.)

- Mixed expressions

```
> (define a 'Bob)
> (define b 'Jane)
> (define c '(Jane))
> (define d '(Bob Jane))

> (equal? a (my-second d))
#f

> (equal? b (my-second d))
#t

> (equal? c (my-second d))
#f
```

Conditional Expressions

- **cond**
  - Evaluates either **true** or **false**

```
> (cond
      [(question) 'outputIfTrue]
      [else 'outputIfFalse])
```

Conditional Expressions (cont.)

- **cond**
  - Evaluates either **true** or **false**

```
> (cond
        [(question) 'outputIfTrue]
        [else 'outputIfFalse])

> (cond
        [(equal? 'a 'a) 'yes]
        [else 'no])
'yes

> (cond
        [(empty? lis) "list is empty"]
        [else "list is not empty"])
"list is empty"
```

Assume "lis" has been defined as an empty list

EXAMPLE 4

- Consider the function that has as input a list parameter named ***lis*** and behaves as follows:
  - If ***lis*** is the *null list*, then the function outputs the **null list**
  - If ***lis*** contains *exactly one data expression*, then the function outputs **a message**
  - If ***lis*** contains *two or more data expressions*, then the function outputs the **second data expression** in ***lis***

EXAMPLE 4 (CONT.)

- Consider the function that has as input a list ***lis*** and behaves as follows:
  - If ***lis*** is the *null list*, then the function outputs the **null list**

```
;; assume lis is a list
> (cond
        [(empty? lis) '()]
        [else ...])
```

Example 4 (cont.)

- Consider the function that has as input a list ***lis*** and behaves as follows:
  - If ***lis*** is the *null list*, then the function outputs the **null list**
  - If ***lis*** contains *exactly one data expression*, then the function outputs a **message**

```
;; assume lis is a list
> (cond
        [(empty? lis) '()]
        [(empty? (rest lis)) "Only one element"]
        [else ...])
```

EXAMPLE 4 (CONT.)

- Consider the function that has as input a list ***lis*** and behaves as follows:
  - If ***lis*** is the *null list*, then the function outputs the **null list**
  - If ***lis*** contains *exactly one data expression*, then the function outputs a **message**
  - If ***lis*** contains *two or more data expressions*, then the function outputs the **second data expression** in ***lis***

```
;; assume lis is a list
> (cond
        [(empty? lis) '()]
        [(empty? (rest lis)) "Only one element"]
        [else (first (rest lis))])
```

Example 4 (cont.)

- Combine all three and we have:

```
> (cond
      [(empty? lis) '()]
      [(empty? (rest lis)) "Only one element"]
      [else (first (rest lis))])
```

Example 4 (cont.)

- We make it a ***function***:

```
> (define this-function
    (lambda (lis)
      (cond
        [(empty? lis) '()]
        [(empty? (rest lis)) "Only one element"]
        [else (first (rest lis))])))
```

- This is the function we saw earlier, ***my-second***, with the difference that this one is handling two possible errors:
  1. empty lists
  2. lists that have only one element

  (we could also add **list?** to check whether it is a list or an atom)

Example 4 (cont.)

- And we test it:

```
> (define a '())
> (define b '(1))
> (define c '(1 2 3))
> (this-function a)
'()
> (this-function b)
"Only one element"
> (this-function c)
2
```

Logical Operators

- **and**
  - Corresponds to &&
- **or**
  - Corresponds to ||
- **not**
  - Corresponds to !

Logical Operators

- Logical operators are also represented in **prefix** form:

```
> (and (equal? 'a 'a) (equal? 'b 'b))
#t

> (and (equal? 'a 'a) (equal? 'a 'b))
#f

> (or (equal? 'a 'a) (equal? 'a 'b))
#t

> (not (equal? 'a 'b))
#t
```

Example 5

- Going back to our function, we add **list?**

```
> (define this-function
    (lambda (lis)
      (cond
        [(not (list? lis)) "Not a list"]
        [(empty? lis) '()]
        [(empty? (rest lis)) "Only one element"]
        [else (first (rest lis))])))
> (this-function 3)
"Not a list"
```

Recursion

- A **recursive function** is a function that includes a call to itself
- A recursive function contains two parts:
  - **Base case** (**trivial case**), which defines the **stopping point** (can have more than one)
  - **Recursive step**, which applies the function to a simpler problem
- You have already seen recursion in **C++**
  - Same concept is applied to **Racket**

Example 1 – Sum of the Squares

- Suppose you want to sum up the **first *n* squares**

```
n = 4

(4*4)+(3*3)+(2*2)+(1*1) = 30
 16   +   9   +   4   +   1    = 30
```

- **Base case** → $n = 1$
- **Recursive steps**

30	= (4 • 4) + sum-squares (n - 1)
14	= (3 • 3) + sum-squares (n - 1)
5	= (2 • 2) + sum-squares (n - 1)
1	= (1 • 1) + sum-squares (n - 1)
n	**= (*n* • *n*) + sum-squares (*n* - 1)**

Example 1 – Sum of the Squares (cont.)

- Our recursive function will have
  - **Base case** → $n = 1 \rightarrow 1$
  - **Recursion** → $(n \cdot n)$ + sum-squares $(n - 1)$

```
(define sum-squares
  (lambda (n)
    (cond
      [(equal? n 1) 1]
      [else (+ (* n n) (sum-squares (- n 1)))])))
```

Example 2 - Factorial

- $4! = 4 \cdot 3 \cdot 2 \cdot 1$

- **Base case** → $n = 0$ $\quad 0! = 1$

- **Recursive steps** → $\quad 4! = 4 \cdot 3!$

$3! = 3 \cdot 2!$

$2! = 2 \cdot 1!$

$1! = 1 \cdot 0!$

$\boldsymbol{n! = n \cdot (n - 1)!}$

Example 2 – Factorial (cont.)

- Our recursive function will have
  - Base case → $n = 0$ → 1
  - Recursion → $(n \cdot \text{factorial}\ (n - 1))$

```
(define factorial
   (lambda (n)
      (cond
        [(equal? n 0) 1]
        [else (* n (factorial (- n 1)))])))
```

Helper Function

- At times you will need to create **helper functions**
- **Example:**
  - Suppose you need to write a recursive function **lili?** (list of lists)
    - **Input:** any list
    - **Output:**

      true if the list is an empty list ***or*** contains **no** atoms,

      false otherwise

```
lili? '()              ➔ #t
lili? '(() 1 ((2)))    ➔ #f
lili? '(() (1) ((2)))  ➔ #t
```

Helper Functions (cont.)

- Our **lili?** function will look like this…

```
(define lili?
   (lambda (lis)
     (cond
       [(empty? lis) true]
       [(atom? (first lis)) false]
       [else (lili? (rest lis))])))
```

Helper Functions (cont.)

- Our **lili?** function will look like this…

```
(define lili?
   (lambda (lis)
     (cond
       [(empty? lis) true]
       [(atom? (first lis)) false]
       [else (lili? (rest lis))])))
```

But you do not have a function atom? defined…

Helper Functions (cont.)

- Our **lili?** function will look like this…

```
(define lili?
   (lambda (lis)
     (cond
       [(empty? lis) true]
       [(atom? (first lis)) false]
       [else (lili? (rest lis))])))
```

```
(define atom?
   (lambda (b)
     (cond
       [(list? b) false]
       [else true])))
```

Another Recursion Example

- Given a **list**, output **true** if the **first element** contains **no atoms**, false otherwise.

Example:

```
(no-atom '())                    => #t
(no-atom '(a))                   => #f
(no-atom '((((()))))             => #t
(no-atom '((((a))))              => #f
(no-atom '((((((a))))))          => #f
(no-atom '((()) b ((c))))        => #t
```

Another Recursion Example (cont.)

- Our recursive function will have
  - Base case → *list is empty* → `#t`
  - Base case → *list has an atom* → `#f`
  - Recursive case → *send back the first element*

We will need the helper function **`atom?`**

```
(define no-atom
  (lambda (lis)
    (cond
      ((empty? lis) #t)
      ((atom? lis) #f)
      (else (no-atom (first lis))))))
```

Possible Incorrect Output

- If you are cons-ing incorrectly, you might get an output that looks a list, BUT it is not.

`(1 2 3)` is NOT equivalent to `'(1 2 . 3)`

- The **dot** indicates a **pair**, not a list of numerical literals.

Racket 2 (end)