- **Evaluation Rules for `cond`**
- **Design Recipe with `cond`**
- **Helper Functions and Reuse**
- **Compound Data**

Recap: Conditionals in Racket

```
(cond
 [question answer]
 ...
 [question answer])
```

- Any number of **`cond`** "lines"
- Each line has one *question* expression and one *answer* expression

```
(define (absolute x)
  (cond
   [(> x 0) x]
   [else (- x)]))
```

```
(absolute 10) → 10

(absolute -7) → 7
```

Evaluation Rules for cond

First question is literally `#true` or `else`

```
(cond
 [#true answer]
 ...
 [question answer])
```

→ *answer*

- Keep only the first answer

Evaluation Rules for cond

First question is literally `#true` or `else`

```
(cond
 [#true answer]                    → answer
 ...
 [question answer])
```

- Keep only the first answer

Example:

```
(+ 1 (cond                → (+ 1 1) → 2
       [#true 1]
       [#false 0]))
```

Evaluation Rules for cond

First question is literally `#true` or `else`

```
(cond
 [#true answer]                    → answer
 ...
 [question answer])
```

- Keep only the first answer

Example:

```
(- 1 (cond                         → (- 1 0) → 1
       [#true 0]
       [(< 10 12) 10]
       [(>= 10 12) 12]))
```

Evaluation Rules for cond

First question is literally `#true` or `else`

```
(cond
 [#true answer]                    → answer
 ...
 [question answer])
```

- Keep only the first answer

Example:

```
(* 1 (cond          → (* 1 0) → 0
      [#true 0]))
```

Evaluation Rules for cond

First question is literally `#false`

```
(cond
 [#false answer]
 [question answer]
 ...
 [question answer])
```

→

```
(cond
 [question answer]
 ...
 [question answer])
```

- Throw away the first line

Evaluation Rules for cond

First question is literally `#false`

```
(cond
 [#false answer]
 [question answer]
 ...
 [question answer])
```

→

```
(cond
 [question answer]
 ...
 [question answer])
```

- Throw away the first line

Example:

```
(+ 1 (cond
       [#false 1]
       [#true 17]))
```

→

```
(+ 1 (cond
       [#true 17]))
```

→ `(+ 1 17)` → `18`

Evaluation Rules for cond

First question isn't a value, yet

```
(cond                          (cond
 [question answer]      →       [nextques answer]
 ...                            ...
 [question answer])             [question answer])
```

where *question* → *nextques*

- Evaluate first question as sub-expression

Evaluation Rules for cond

First question isn't a value, yet

```
(cond                          (cond
 [question answer]      →       [nextques answer]
 ...                            ...
 [question answer])             [question answer])
```

where *question* → *nextques*

- Evaluate first question as sub-expression

Example:

```
(+ 1 (cond          → (+ 1 (cond
       [(< 1 2) 5]           [#true 5]
       [else 8]))            [else 8]))

                    → (+ 1 5) → 6
```

Evaluation Rules for cond

No true answers

```
(cond)
```
→ *error*

Just an `else`

```
(cond
 [else answer])
```
→ *answer*

- **Evaluation Rules for `cond`**
- **Design Recipe with `cond`**
- **Helper Functions and Reuse**
- **Compound Data**

Design Recipe I

Data

- Understand the input data: `num`, `bool`, `string`, or `image`

Signature, Purpose, and Header

- Describe (but don't write) the function

Examples

- Show what will happen when the function is done

Body

- The most creative step: implement the function body

Test

- Run the examples

Examples

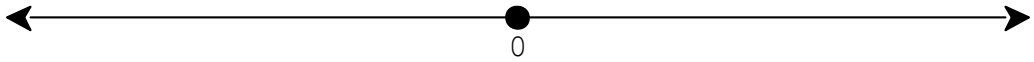
When the problem statement divides the input into several categories, test each one

Examples

When the problem statement divides the input into several categories, test each one

Example:

Write the function `line-part` that determines whether a number is on zero, to the left, or to the right on a number line

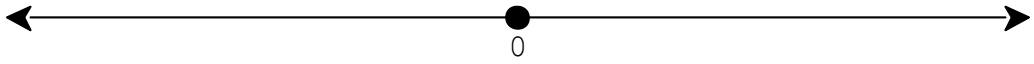


Examples

When the problem statement divides the input into several categories, test each one

Example:

Write the function `line-part` that determines whether a number is on zero, to the left, or to the right on a number line




```
(check-expect (line-part 0) "zero")
(check-expect (line-part -3) "left")
(check-expect (line-part 3) "right")
```

Design Recipe I

Data

- Understand the input data: `num`, `bool`, `string`, or `image`

Signature, Purpose, and Header

- Describe (but don't write) the function

Examples

- Show what will happen when the function is done

Body

- The most creative step: implement the function body

Test

- Run the examples

Body

When the problem statement divides the input into *N* categories:

- Start the body with a **`cond`** expression and *N* lines
- Formulate a question to recognize each category

Body

When the problem statement divides the input into *N* categories:

- Start the body with a **`cond`** expression and *N* lines
- Formulate a question to recognize each category

Example:

Write the function **`line-part`** that determines whether
a number is on zero, to the left, or to the right on a number line

Body

When the problem statement divides the input into *N* categories:

- Start the body with a **`cond`** expression and *N* lines
- Formulate a question to recognize each category

Example:

Write the function **`line-part`** that determines whether
a number is on zero, to the left, or to the right on a number line

Three cases, so three lines:

```
(define (line-part n)
  (cond
   [(= n 0) ...]
   [(< n 0) ...]
   [(> n 0) ...]))
```

- **Evaluation Rules for `cond`**
- **Design Recipe with `cond`**
- **Helper Functions and Reuse**
- **Compound Data**

Designing Programs

Design recipe

- As outlined last lecture

Designing Programs

Design recipe

- As outlined last lecture

Helper functions and reuse

- Writing writing a function, consider whether existing functions help
  - Example: `insert-at-middle` uses `middle`
- Look for functions that you wish you had written
  - Example: `same-person-maybe-disguised?` needs `wearing-beard?`

Another Example

Write the function **`bigger-image?`** which checks whether one image has more pixels than a second image

Another Example

Write the function **`bigger-image?`** which checks whether one image has more pixels than a second image

```
; image image -> bool
```

Another Example

Write the function **`bigger-image?`** which checks whether one image has more pixels than a second image

```
; image image -> bool
; Returns true if a has more pixels than b
```

Another Example

Write the function **bigger-image?** which checks whether one image has more pixels than a second image

```
; image image -> bool
; Returns true if a has more pixels than b
(define (bigger-image? a b) ...)
```

Another Example

Write the function **bigger-image?** which checks whether one image has more pixels than a second image

```
; image image -> bool
; Returns true if a has more pixels than b
(define (bigger-image? a b) ...)
```

```
(check-expect (bigger-image? ■ ■) #true)
(check-expect (bigger-image? ■ ■) #false)
```

Another Example

Write the function **bigger-image?** which checks whether one image has more pixels than a second image

```
; image image -> bool
; Returns true if a has more pixels than b
(define (bigger-image? a b)
  (> (* (image-width a) (image-height a))
     (* (image-width b) (image-height b))))

(check-expect (bigger-image? ■ ■) #true)
(check-expect (bigger-image? ■ ■) #false)
```

Another Example

Write the function **bigger-image?** which checks whether one image has more pixels than a second image

```
; image image -> bool
; Returns true if a has more pixels than b
(define (bigger-image? a b)
  (> (image-size a) (image-size b)))


(check-expect (bigger-image? ■ ■) #true)
(check-expect (bigger-image? ■ ■) #false)
```

Wish list: **image-size**

Another Example

Write the function **`bigger-image?`** which checks whether one image has more pixels than a second image

```
; image image -> bool
; Returns true if a has more pixels than b
(define (bigger-image? a b)
  (> (image-size a) (image-size b)))
```

```
(check-expect (bigger-image? ■ ■) #true)
(check-expect (bigger-image? ■ ■) #false)
```

Wish list: **`image-size`**

Fullfill wishes by applying the recipe again

(exercise for the reader)

Reuse

We should be able to use **`bigger-image?`** to write the **`max-image`** function

Reuse

We should be able to use **bigger-image?** to write the **max-image** function

```
; image image -> image
; Returns a if a has more pixels than b,
; otherwise returns b
(define (max-image a b) ...)
```

Reuse

We should be able to use **`bigger-image?`** to write the **`max-image`** function

```
; image image -> image
; Returns a if a has more pixels than b,
; otherwise returns b
(define (max-image a b) ...)
```

```
(check-expect (max-image ■ ■) ■)
(check-expect (max-image ■ ■) ■)
```

Reuse

We should be able to use **bigger-image?** to write the **max-image** function

```
; image image -> image
; Returns a if a has more pixels than b,
; otherwise returns b
(define (max-image a b)
  ... (bigger-image? a b) ...)
```

```
(check-expect (max-image ■ ■) ■)
(check-expect (max-image ■ ■) ■)
```

Reuse

We should be able to use **bigger-image?** to write the **max-image** function

```
; image image -> image
; Returns a if a has more pixels than b,
; otherwise returns b
(define (max-image a b)
  (cond
   [(bigger-image? a b) a]
   [else b]))

(check-expect (max-image ■ ■) ■)
(check-expect (max-image ■ ■) ■)
```

- **Evaluation Rules for `cond`**
- **Design Recipe with `cond`**
- **Helper Functions and Reuse**
- **Compound Data**

Positions

- A **`posn`** is

  `(make-posn X Y)`

  where **`X`** is a **`num`** and **`Y`** is a **`num`**

Positions

- A **`posn`** is

  `(make-posn X Y)`

  where **`X`** is a **`num`** and **`Y`** is a **`num`**

Examples:

```
(make-posn 1 2)

(make-posn 17 0)
```

Positions

- A **posn** is

  (make-posn X Y)

  where **X** is a **num** and **Y** is a **num**

Examples:

```
(make-posn 1 2)

(make-posn 17 0)
```

A **posn** is a value, just like a number, symbol, or image

posn-x and posn-y

The **posn-x** and **posn-y** operators extract numbers from a **posn**:

```
(posn-x (make-posn 1 2)) → 1

(posn-y (make-posn 1 2)) → 2
```

posn-x and posn-y

The **posn-x** and **posn-y** operators extract numbers from a **posn**:

```
(posn-x (make-posn 1 2)) → 1

(posn-y (make-posn 1 2)) → 2
```

- General evaluation rules for any values **X** and **Y**:

```
(posn-x (make-posn X Y)) → X

(posn-y (make-posn X Y)) → Y
```

Positions and Values

Is `(make-posn 100 200)` a value?

Positions and Values

Is `(make-posn 100 200)` a value?

Yes.

A `posn` is

`(make-posn X Y)`

where `X` is a `num` and `Y` is a `num`

Positions and Values

Is `(make-posn (+ 1 2) 200)` a value?

Positions and Values

Is `(make-posn (+ 1 2) 200)` a value?

No. `(+ 1 2)` is not a `num`, yet.

Positions and Values

Is `(make-posn (+ 1 2) 200)` a value?

No. `(+ 1 2)` is not a `num`, yet.

- Two more evaluation rules:

```
(make-posn X Y) → (make-posn Z Y)
                          when X → Z

(make-posn X Y) → (make-posn X Z)
                          when Y → Z
```

Positions and Values

Is `(make-posn (+ 1 2) 200)` a value?

No. `(+ 1 2)` is not a `num`, yet.

- Two more evaluation rules:

`(make-posn X Y)` → `(make-posn Z Y)`
when `X` → `Z`

`(make-posn X Y)` → `(make-posn X Z)`
when `Y` → `Z`

Example:

```
(make-posn (+ 1 2) 200) →
(make-posn 3 200)
```

More Examples

Try these in DrRacket's stepper:

```
(make-posn (+ 1 2) (+ 3 4))

(posn-x (make-posn (+ 1 2) (+ 3 4)))

; posn -> num
(define (pixels-from-corner p)
  (+ (posn-x p) (posn-y p)))
(pixels-from-corner (make-posn 1 2))

; posn -> posn
(define (flip p)
  (make-posn (posn-y p) (posn-x p)))
(flip (make-posn 1 2))
```