

## Outline

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

## Designing Programs

### Design recipe

- As outlined last lecture

### Helper functions and reuse

- Writing writing a function, consider whether existing functions help
  - Example: `wearing-glasses?` uses `add-glasses`
- 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

```
; bigger-image? : image image -> bool
```

## Another Example

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

```
; bigger-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

```
; bigger-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

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

```
(bigger-image?  ) "should be" true
(bigger-image?  ) "should be" false
```

## Another Example

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

```
; bigger-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))))
```

```
(bigger-image?  ) "should be" true
(bigger-image?  ) "should be" false
```

## Another Example

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

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

```
(bigger-image? ■ ■) "should be" true
(bigger-image? ■ ■) "should be" false
```

*Wish list:* `image-size`

## Another Example

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

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

```
(bigger-image? ■ ■) "should be" true
(bigger-image? ■ ■) "should be" 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

```
; max-image : 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

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

(max-image ■ ■) "should be" ■
(max-image ■ ■) "should be" ■
```

## Reuse

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

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

(max-image ■ ■) "should be" ■
(max-image ■ ■) "should be" ■
```

## Reuse

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

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

(max-image ■ ■) "should be" ■
(max-image ■ ■) "should be" ■
```

Instead of returning a `bool`, we need to do one of two things, so we need `cond`

## Outline

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

## Conditionals in Algebra

General format of conditionals in algebra:

$$\left\{ \begin{array}{ll} \text{answer} & \text{question} \\ \dots & \\ \text{answer} & \text{question} \end{array} \right.$$

Example:

$$\text{abs}(x) = \begin{cases} x & \text{if } x > 0 \\ -x & \text{otherwise} \end{cases}$$

$$\text{abs}(10) = 10$$

$$\text{abs}(-7) = 7$$

## Conditionals

General syntax of `cond` in our language:

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

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

```
(define (abs x)
  (cond
    [(> x 0) x]
    [else (- x)]))
(abs 10) "should be" 10
(abs -7) "should be" 7
```

## Completing max-image

- Use `cond` to complete `max-image`

```
(define (max-image a b)
  (cond
    [(bigger-image? a b) a]
    [else b]))
```

## Outline

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

## 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
 [true 0])) → (* 1 0) → 0
```

## Evaluation Rules for cond

First question is literally `false`

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

- Throw away the first line

Example:

```
(+ 1 (cond
 [false 1] → (+ 1 (cond
 [true 17]))
 [true 17]))
 → (+ 1 17) → 18
```

## Evaluation Rules for cond

First question isn't a value, yet

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

where `question` → `nextques`

- Evaluate first question as sub-expression

Example:

```
(+ 1 (cond
 [(< 1 2) 5] → (+ 1 (cond
 [true 5]
 [else 8]))
 [else 8]))
 → (+ 1 5) → 6
```

## Evaluation Rules for cond

No true answers

```
(cond) → error
```

## Outline

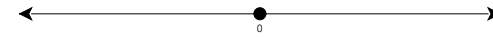
- ▶ Helper Functions and Reuse
- ▶ Conditionals
- ▶ Evaluation Rules for `cond`
- ▶ Design Recipe with `cond`
- ▶ Compound Data

## 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



```
(line-part 0) "should be" 'zero  
(line-part -3) "should be" 'left  
(line-part 3) "should be" 'right
```

## 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) ...]))
```

## Outline

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

## Finding Images



## Image Tests in Conditionals

Now we can combine such operators with `cond`:

```
; detect-person : image image image -> image  
; Returns a or b, depending on which is in i  
(define (detect-person i a b)  
  (cond  
    [(image-inside? i a) a]  
    [(image-inside? i b) b]))
```



## Finding and Adjusting Images

Suppose we want to write `frame-person`:



## Finding an Image Position

~~`find-image : image image -> num num`~~

**Must return a single value**

Correct contract:

`find-image : image image -> posn`

- A `posn` is a *compound value*

Need an operator that reports *where* an image exists



## 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
```

- General evaluation rules for any `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?

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?

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 DrScheme's stepper:

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

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

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

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