.helper Functions and Reuse

- 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: `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 \texttt{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

```scheme
; bigger-image? : image image --> bool
```
Another Example

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

```scheme
; 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

```scheme
; 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

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

```scheme
(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

```scheme
; 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))))

(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

`; 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)))

(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

```scheme
; 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)))
```

```scheme
(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 \texttt{bigger-image?} to write the \texttt{max-image} function

\begin{verbatim}
; max-image : image image \rightarrow image
; Returns a if a has more pixels than b,
; otherwise returns b
(define (max-image a b) ...) 
\end{verbatim}
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) ...)

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

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

```scheme
; 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) ...)
```

```scheme
(check-expect (max-image ■ ■) ■)
(check-expect (max-image ■ □) □)
```
Reuse

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

```scheme
; 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) ...)
```

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

Instead of returning a `bool`, we need to do one of two things, so we need `cond`
Recap: Conditionals in Racket

\[
(\text{cond} \\
[\text{question answer}] \\
\ldots \\
[\text{question answer}])
\]

• Any number of \texttt{cond} “lines”

• Each line has one \textit{question} expression and one \textit{answer} expression

\[
(\text{define} \ (\text{absolute x}) \\
(\text{cond} \\
[(> x 0) x] \\
[\text{else} (- x)])))
\]

\[
(\text{absolute 10}) \rightarrow 10 \\
(\text{absolute -7}) \rightarrow 7
\]
Completing max-image

Use **cond** to complete **max-image**: 

```
(define (max-image a b)
  (cond
    [(bigger-image? a b) a]
    [else b]))
```
Helper Functions and Reuse

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 → (* 1 0) → 0
  [true 0]))
```
Evaluation Rules for cond

First question is literally $\text{false}$

$$(\text{cond}$$
  $$\text{false answer}$$
  $$\text{question answer}$$
  $$\ldots$$
  $$\text{question answer}$$)

$$(\text{cond}$$
  $$\text{question answer}$$
  $$\ldots$$
  $$\text{question answer}$$))

- Throw away the first line

Example:

$$(\text{cond}$$
  $$\text{false 1}$$
  $$\text{true 17}$$
  $$\text{true 17}$$)

$$(\text{cond}$$
  $$\text{true 17}$$)

$$(+ 1 17) \rightarrow 18$$
Evaluation Rules for cond

First question isn’t a value, yet

\[
\begin{align*}
\text{(cond} & \quad \text{(cond)} \\
\text{[question answer]} & \quad \text{[nextques answer]} \\
\ldots & \quad \ldots \\
\text{[question answer]} & \quad \text{[question answer]}) \\
\end{align*}
\]

where \text{question} \rightarrow \text{nextques}

- Evaluate first question as sub-expression

Example:

\[
\begin{align*}
(+ 1 \ (\text{cond} & \quad (+ 1 \ (\text{cond}) \\
\quad [\langle 1 2 \rangle 5] & \quad [\text{true 5}] \\
\quad [\text{else 8}]) & \quad [\text{else 8}]))) \\
\rightarrow & \quad (+ 1 5) \rightarrow 6
\end{align*}
\]
Evaluation Rules for cond

No true answers

\((\text{cond}) \rightarrow \text{error}\)
- Helper Functions and Reuse
- 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

![Number line](image)

```
(check-expect (line-part 0) "zero")
(check-expect (line-part -3) "left")
(check-expect (line-part 3) "right")
```
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) ...]))
```
Helper Functions and Reuse

Evaluation Rules for cond

Design Recipe with cond

Compound Data
Dropping a Ball

Last time, we wrote **distance**: 

```scheme
; distance : num [positive] -> num
; Takes t seconds and report how far
; a ball has fallen in feet at that time
(define (distance t)
  (* 1/2 32 (expt t 2)))

(check-expect (distance 0) 0)
(check-expect (distance 1) 16)
(check-expect (distance 100) 160000)
```
Tossing a Ball

Suppose that we give the ball an initial velocity:

\[
; \text{toss : num num -> num} \\
; \text{With initial velocity v0 after t seconds,} \\
; \text{how far a ball is from its initial position} \\
\text{(define (toss v0 t)} \\
\quad (+ (* v0 t) (distance t)))
\]

(check-expect (toss 10 0) 0)
(check-expect (toss 10 1) 26)
(check-expect (toss -10 1) 6)
(check-expect (toss -32 1) -16)
Tossing a Ball Sideways

Suppose that we don’t toss the ball straight up:

; toss-2d : num num num -> num num

Must return a single value

Correct contract:

; toss-2d : num num num -> posn

A posn is a compound value
Positions

• A **posn** is

  \[(\texttt{make-posn } \texttt{X} \texttt{ Y})\]

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

Examples:

\[(\texttt{make-posn } 1 \texttt{ 2})\]

\[(\texttt{make-posn } 17 \texttt{ 0})\]

A **posn** is a value, just like a number, symbol, or image
posn-x and posn-y

The \texttt{posn-x} and \texttt{posn-y} operators extract numbers from a \texttt{posn}:

\[
(posn-x \ (make-posn \ 1 \ 2)) \rightarrow 1
\]

\[
(posn-y \ (make-posn \ 1 \ 2)) \rightarrow 2
\]

• General evaluation rules for any \( X \) and \( Y \):

\[
(posn-x \ (make-posn \ X \ Y)) \rightarrow X
\]

\[
(posn-y \ (make-posn \ X \ Y)) \rightarrow Y
\]
Positions and Values

Is \(\text{make-posn 100 200}\) a value?

Yes.

A \textit{posn} is

\(\text{make-posn X Y}\)

where \(X\) is a \textit{num} and \(Y\) is a \textit{num}
Positions and Values

Is \((\text{make-posn} \ (+ \ 1 \ 2) \ 200)\) a value?

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

• Two more evaluation rules:

\[
(\text{make-posn} \ X \ Y) \rightarrow (\text{make-posn} \ Z \ Y)
\]
when \(X \rightarrow Z\)

\[
(\text{make-posn} \ X \ Y) \rightarrow (\text{make-posn} \ X \ Z)
\]
when \(Y \rightarrow Z\)

Example:

\[
(\text{make-posn} \ (+ \ 1 \ 2) \ 200) \rightarrow
(\text{make-posn} \ 3 \ 200)
\]
More Examples

Try these in DrRacket’s stepper:

\[(\text{make-posn } (+ \ 1 \ 2) \ (+ \ 3 \ 4))\]

\[(\text{posn-x } (\text{make-posn } (+ \ 1 \ 2) \ (+ \ 3 \ 4)))\]

; 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))