Evaluation Rules for cond

- Design Recipe with cond
- Helper Functions and Reuse
- Compound Data
Recap: Conditionals in Racket

\[
\text{(cond }
\begin{array}{l}
\text{[question answer]} \\
\ldots \\
\text{[question answer]}) \\
\end{array}
\]

- Any number of \text{cond} “lines”
- Each line has one \text{question} expression and one \text{answer} expression

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

\[
(\text{absolute 10)} \rightarrow 10 \\
(\text{absolute -7)} \rightarrow 7
\]
Evaluation Rules for cond

First question is literally **true** or **else**

\[
\text{(cond}
\begin{align*}
\text{[true answer]} & 
\rightarrow \text{answer} \\
\text{...} \\
\text{[question answer]} & 
\end{align*}
\]

• Keep only the first answer

Example:

\[
(* 1 \text{ (cond} \rightarrow (* 1 0) \rightarrow 0
\begin{align*}
\text{[true 0]} & 
\end{align*}
\)\]
Evaluation Rules for cond

First question is literally false

\[
\text{(cond}
  \begin{array}{l}
\text{[false answer]}
\text{[question answer]}
\text{[question answer]}
\end{array}
\rightarrow
\text{(cond}
  \begin{array}{l}
\text{[question answer]}
\text{[question answer]}
\end{array}
\]

\bullet \text{ Throw away the first line}

Example:

\[
(+ \ 1 \ (\text{cond}
  \begin{array}{l}
\text{[false 1]}
\text{[true 17]}
\end{array}
\rightarrow
(+ \ 1 \ (\text{cond}
  \begin{array}{l}
\text{[false 1]}
\text{[true 17]}
\end{array}

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

First question isn’t a value, yet

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

where question → nextques

• Evaluate first question as sub-expression

Example:

\[
(+ 1 \ (\text{cond} \\
\quad \text{[(< 1 2) 5]} \\
\quad \text{[else 8]])} \)) \rightarrow (+ 1 \ (\text{cond} \\
\quad \text{[true 5]} \\
\quad \text{[else 8]])}) \rightarrow (+ 1 5) \rightarrow 6
\]
Evaluation Rules for cond

No true answers

\[(\text{cond}) \rightarrow \text{error}\]

Just an \textbf{else}

\[(\text{cond}
\quad \text{[else } \text{answer}]\) \rightarrow \text{answer}\]
Evaluation Rules for $\text{cond}$

Design Recipe with $\text{cond}$

Helper Functions and Reuse

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

```
(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:

```scheme
(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

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 $\text{bigger-image?}$ 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.

```scheme
; bigger-image? : image image -> bool
; Returns true if a has more pixels than b
```
Another Example

Write the function \texttt{bigger-image?} which checks whether one image has more pixels than a second image

\texttt{; bigger-image? : image image \rightarrow bool}
\texttt{; Returns true if \texttt{a} has more pixels than \texttt{b}}
\texttt{(define \texttt{(bigger-image? a b)} \ldots)}
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) ...)

(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-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

```
; 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*

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) ...)

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

(check-expect (max-image ■ ■) ■)
(check-expect (max-image ■ ■) ■)
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)
  (cond
   [(bigger-image? a b) a]
   [else b]))

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

Convert streets in the Avenues to blocks east of Main
Avenues

Convert streets in the Avenues to blocks east of Main

A St. = 260 E
...
O St. = 1000 E
...

• A street at 2.6
• 10 - 2.6 blocks in 14 streets

1 St. = 682.85... E
Avenues

; street->slc : string -> num
; Converts streets to blocks east of Main:
(define (street->slc st)
   (+ 2.6 (* (street-index st)
             (/ (- 10 2.6) 14)))))

; street-index : string -> num
; Converts "A" to 0, "B" to 1, etc.
(define (street-index st)
   (- (char->integer (string-ref st 0))
     (char->integer #\A))
(check-expect (street-index "A") 0)
(check-expect (street-index "O") 14)

(check-expect (street->slc "A") 2.6)
(check-within (street->slc "I") 6.83 0.01)
(check-expect (street->slc "O") 10)
➤ Evaluation Rules for cond
➤ Design Recipe with cond
➤ Helper Functions and Reuse
➤ Compound Data
Transforming a Point

Convert Avenues corners to SLC coordinates

; ave->slc : string num --> num num

Must return a single value

Correct contract:

; ave->slc : string num --> posn

A posn is a compound value
Positions

- A \texttt{posn} is

\begin{center}
\texttt{(make-posn X Y)}
\end{center}

where \texttt{X} is a \texttt{num} and \texttt{Y} is a \texttt{num}

Examples:

\begin{center}
\texttt{(make-posn 1 2)}
\end{center}

\begin{center}
\texttt{(make-posn 17 0)}
\end{center}

A \texttt{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:

\[(\text{posn-x } (\text{make-posn } 1 \ 2)) \rightarrow 1\]
\[(\text{posn-y } (\text{make-posn } 1 \ 2)) \rightarrow 2\]

• General evaluation rules for any values X and Y:
\[(\text{posn-x } (\text{make-posn } X \ Y)) \rightarrow X\]
\[(\text{posn-y } (\text{make-posn } X \ Y)) \rightarrow Y\]
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) \rightarrow (make-posn Z Y)`
  
  when \( X \rightarrow Z \)

  `(make-posn X Y) \rightarrow (make-posn X Z)`
  
  when \( Y \rightarrow Z \)

Example:

`(make-posn (+ 1 2) 200) \rightarrow`

`(make-posn 3 200)`
More Examples

Try these in DrRacket’s stepper:

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

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

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

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