Module 4: Making Decisions

If you have not already, make sure you

- Read *How to Design Programs*, Section 4
Booleans (Bool)

<, >, <=, >=, =, and equal? are new functions, each of which returns a boolean value (Bool).

\[
\begin{align*}
(< 4 \ 6) & \iff 4 < 6 \\
(> 4 \ 6) & \iff 4 > 6 \\
(= 5 \ 7) & \iff 5 = 7 \\
(\geq 5 \ 5) & \iff 6 \geq 5 \\
(\leq 5 \ 5) & \iff 5 \leq 5
\end{align*}
\]

Each returns #true or #false. These are the only values a Bool may take.

(You may see #true called true or #t, and see #false called false or #f.)

A function which returns a Bool is called a predicate. Often the name of predicates end with ?, as in string=?.

Other predicates include even?, odd?, and equal?.
A sin-squared window, used in signal processing, can be described by

\[
f(x) = \begin{cases} 
0 & \text{for } x < 0 \\
1 & \text{for } x \geq 1 \\
sin^2(x\pi/2) & \text{for } 0 \leq x < 1 
\end{cases}
\]

Racket gives us an easy way to design such things in a special form called cond.
Using *cond* to describe cases

\[
 f(x) = \begin{cases} 
 0 & \text{for } x < 0 \\
 1 & \text{for } x \geq 1 \\
 \sin^2(x\pi/2) & \text{for } 0 \leq x < 1 
\end{cases}
\]

\[
\text{(define (ssqw x)} \\
\text{(cond} \\
\text{[(< x 0) 0]} \\
\text{[(>= x 1) 1]} \\
\text{[(< x 1) (sqr (sin (* x pi 0.5)))]])}
\]

*cond* is a special form, not a function. We deal with it in a special way. In particular, *do not evaluate its arguments* until necessary.

Each argument of *cond* is a pair of square brackets around a pair of expressions: [question answer].
Evaluating a cond statement

How do we evaluate a cond?

**Informally**, evaluate a cond by considering the question/answer pairs in order, top to bottom. When considering a question/answer pair, evaluate the question. If the question evaluates to **#true**, the whole cond returns the answer.

```lisp
(define (ssqw x)
  (cond
    [(< x 0) 0]
    [(>= x 1) 1]
    [(< x 1) (sqr (sin (* x pi 0.5)))])
)
```

For example consider, (ssqw 4).

=> (cond [(< 4 0) 0]
        [(>= 4 1) 1]
        [(< 4 1) (sqr (sin (* 4 pi 0.5)))])

The first question is **#false**, but the second question is **#true**!

=> 1
What happens if *none* of the questions evaluate to `#true`?

```scheme
(define (absolute-value n)
  (cond
    [(> n 0) n]
    [(< n 0) (- n)]))
```

An error occurs if we try to run `(absolute-value 0)`

This can be helpful — if we try to consider all the possibilities, but we miss one, testing may raise this error. Then we can fix it.

But sometimes we want to only describe some conditions, and do something different if none of them are satisfied.
We could use a question which always evaluates to \( \texttt{#true} \):

\[
\begin{array}{l}
\texttt{(define (absolute-value n)} \\
\texttt{\quad (cond)} \\
\texttt{\quad \quad [(> n 0) n]} \\
\texttt{\quad \quad [\texttt{#true} (- n)]])}
\end{array}
\]

Remember: the question/answer pairs are considered \emph{in order}, top to bottom, and it stops as soon as it finds a question which evaluates to \texttt{#true}.

This is useful sufficiently frequently that there is special keyword for it: \texttt{else}.

\[
\begin{array}{l}
\texttt{(define (absolute-value n)} \\
\texttt{\quad (cond)} \\
\texttt{\quad \quad [(> n 0) n]} \\
\texttt{\quad \quad [\texttt{else} (- n)]])}
\end{array}
\]
Tracing cond

Recall we are imagining interpreting our programs as a series of substitutions, called a trace. How do we formally trace cond?

The general form of a conditional is

\[
\text{cond} \quad \text{[question1 answer1]} \\
\text{[question2 answer2]} \\
\ldots \\
\text{[questionk answerk]}
\]

To evaluate the conditional, evaluate question1, then perform the following substitutions:

- \((\text{cond} \quad [#false \quad \text{exp0}][\text{exp1 exp2}]\ldots) \Rightarrow (\text{cond} \quad [\text{exp1 exp2}]\ldots)\)
- \((\text{cond} \quad [#true \quad \text{exp0}][\text{exp1 exp2}]\ldots) \Rightarrow \text{exp0}\)
- \((\text{cond} \quad [\text{else exp0}]) \Rightarrow \text{exp0}\)
Tracing cond example

- (cond [#false exp0][exp1 exp2]...) ⇒ (cond [exp1 exp2]...)
- (cond [#true exp0][exp1 exp2]...) ⇒ exp0
- (cond [else exp0]) ⇒ exp0

(define (ssqw x) ...)

(ssqw 0)
⇒ (cond [(< 0 0) 0] [(>= 0 1) 1] [(< 0 1) (sqr (sin (* 0 pi 0.5)))]))
⇒ (cond [false 0] [(>= 0 1) 1] [(< 0 1) (sqr (sin (* 0 pi 0.5)))]))
⇒ (cond [(>= 0 1) 1] [(< 0 1) (sqr (sin (* 0 pi 0.5)))]))
⇒ (cond [false 1] [(< 0 1) (sqr (sin (* 0 pi 0.5)))]))
⇒ (cond [(< 0 1) (sqr (sin (* 0 pi 0.5)))]))
⇒ (cond [true (sqr (sin (* 0 pi 0.5)))]))
⇒ (sqr (sin (* 0 pi 0.5)))
⇒ (sqr (sin 0))
⇒ (sqr 0)
⇒ 0
Tracing cond

(define (qux a b)
  (cond
    [(= a b) 42]
    [(> a (+ 3 b)) (* a b)]
    [(> a b) (- b a)]
    [else -42]))

(qux 5 4)

Ex. Perform a complete trace of this program.
You should write tests:
- so each question is evaluated to `#true` at least once, to verify each answer is tested.
- to test boundaries; it is easy to get “off-by-one” errors!

Suppose I wanted a function which returns 0 for integers \( x \leq 1 \), 1 for integers \( 1 \leq x \leq 10 \), and 2 for other integers. What should I test?
I should check boundaries \((-1, 0, 1)\) and \((10, 11)\), some other negative number, and some larger number.

```scheme
;; (categorize n) return 0 for negative, 1 for positive <= 10, 2 otherwise.
(define (categorize n)
  (cond
   [(< n 0) 0]
   [(< n 10) 1]
   [else 2]))
```

;; Tests:
(check-expect (categorize -5) 0)
(check-expect (categorize -1) 0)
(check-expect (categorize 1) 1)
(check-expect (categorize 10) 1)
(check-expect (categorize 11) 2)
(check-expect (categorize 50) 2)
(check-expect (categorize 0) 2)
Conditionals can be very useful in combination with map

;;; (fix-limit val) replace val with 20 if it is greater
;;; than 20, and with 10 if it is lower than 10.
;;; fix-limit: Num -> Num
;;; Example:
(check-expect (fix-limit 5) 10)

(define (fix-limit val)
  (cond [(> val 20) 20]
        [(< val 10) 10]
        [else val]))

;;; (fix-list L) Replace each value in L with 20 if it is greater
;;; than 20, and with 10 if it is lower than 10.
;;; fix-list: Num Num (listof Num) -> (listof Num)
;;; Example:
(check-expect (fix-list (list 8 12 18 22)) (list 10 12 18 20))

(define (fix-list L)
  (map fix-limit L))
Conditionals can be very useful in combination with map

;;; (fix-limit val) replace val with 20 if it is greater
;;; than 20, and with 10 if it is lower than 10.
(define (fix-limit val)
   (cond [(> val 20) 20]
         [(< val 10) 10]
         [else val]))

;;; (fix-list L) Replace each value in L with 20 if it is greater
;;; than 20, and with 10 if it is lower than 10.
(define (fix-list L)
   (map fix-limit L))

Using cond and map, write a function that consumes a (listof Int). The function makes all odd numbers negative, and all even numbers positive.
(neg-odd (list 2 5 8 -11 -14 -17)) => (list 2 -5 8 -11 14 -17)
and now for something completely different

We combine predicates using the special forms **and**, **or**, and the function **not**. These all consume and return **Bool** values.

- **and** returns **#false** if at least one of its arguments is **#false**, and **#true** otherwise.
- **or** returns **#true** if at least one of its arguments is **#true** and false otherwise.
- **not** returns **#true** if its argument is **#false**, and **#false** if its argument is **#true**.

A few examples:

- \((\text{and} \ (>) 5 \ 4) \ (>\ 7 \ 2)) => \ #true\)
- \((\text{or} \ (\geq) 5 \ 4) \ (>\ 7 \ 2)) => \ #true\)
- \((\text{and} \ (\geq) 5 \ 5) \ (\leq) 7 \ 2) \ (>\ 5 \ 1)) => \ #false\)
- \((\text{or} \ (>\ 4 \ 5) \ (>\ 2 \ 7) \ (<\ 9 \ 4)) => \ #false\)
- \((\text{not} \ (=\ 5 \ 4)) => \ #true\)
- \((\text{and} \ #true \ (<\ 3 \ 7) \ (\geq) 9 \ 1)) => \ #true\)

Both **or** and **and** require at least two arguments, but may have more.

**Exercise**

Write a function that consumes a **Num**, and returns

- "big" if \(80 < x \leq 100\),
- "small" if \(0 < x \leq 80\),
- "invalid" otherwise.
An important subtlety interpreting `and` and `or`: short-circuiting

`and` and `or` are *not* functions. They are **special forms**. Do not evaluate their arguments until necessary.

Informally, evaluate the arguments one by one, and *stop as soon as possible*.

For example:

```scheme
(define (baz x)
  (and (not (= 0 x))
       (> 0 (cos (/ 1 x)))))
```

If I run `(baz 0)`, attempting to evaluate the expression `(/ 1 x)`, would cause a division by zero error. But when `x` is zero, the first argument of `and` is `#false`, so the second is not evaluated.
Substitution rules for \texttt{and}

Use the following rules for tracing \texttt{and}:

- \[(\texttt{and} \ #\texttt{true} \ \texttt{exp} \ ... ) \Rightarrow (\texttt{and} \ \texttt{exp} \ ... )\]
- \[(\texttt{and} \ #\texttt{false} \ \texttt{exp} \ ... ) \Rightarrow \ #\texttt{false}\]
- \[(\texttt{and}) \Rightarrow \ #\texttt{true}\]

Note: this is not what the stepper does! If in this course you are asked to perform a trace, follow these rules.

Exercise

Perform a trace of

\[(\texttt{and} \ (= 3 \ 3) \ (> 7 \ 4) \ (< 7 \ 4) \ (> 0 \ (/ \ 3 \ 0)))\]

\[
\Rightarrow \ (\texttt{and} \ #\texttt{true} \ (> 7 \ 4) \ (< 7 \ 4) \ (> 0 \ (/ \ 3 \ 0)))
\]

\[
\Rightarrow \ (\texttt{and} \ (> 7 \ 4) \ (< 7 \ 4) \ (> 0 \ (/ \ 3 \ 0)))
\]

\[
\Rightarrow \ (\texttt{and} \ #\texttt{true} \ (< 7 \ 4) \ (> 0 \ (/ \ 3 \ 0)))
\]

\[
\Rightarrow \ (\texttt{and} \ (< 7 \ 4) \ (> 0 \ (/ \ 3 \ 0)))
\]

\[
\Rightarrow \ (\texttt{and} \ #\texttt{false} \ (> 0 \ (/ \ 3 \ 0)))
\]

\[
\Rightarrow \ #\texttt{false}
\]
Substitution rules for \texttt{or}

Use the following rules for tracing \texttt{or}:

\begin{itemize}
  \item \((\texttt{or \ #true \ exp \ ...}) \Rightarrow \ #true\)
  \item \((\texttt{or \ #false \ exp \ ...}) \Rightarrow (\texttt{or \ exp \ ...})\)
  \item \((\texttt{or}) \Rightarrow \ #false\)
\end{itemize}

Note: this is not what the stepper does! If in this course you are asked to perform a trace, follow these rules.

**Exercise**

Perform a trace of

\[(\texttt{or \ (< \ 7 \ 4) \ (= \ 3 \ 3) \ (> \ 7 \ 4) \ (> \ 0 \ (/\ 3 \ 0)))}\]

\[\Rightarrow \ (\texttt{or \ #false \ (= \ 3 \ 3) \ (> \ 7 \ 4) \ (> \ 0 \ (/\ 3 \ 0)))}\]

\[\Rightarrow \ (\texttt{or \ (= \ 3 \ 3) \ (> \ 7 \ 4) \ (> \ 0 \ (/\ 3 \ 0)))}\]

\[\Rightarrow \ (\texttt{or \ #true \ (> \ 7 \ 4) \ (> \ 0 \ (/\ 3 \ 0)))}\]

\[\Rightarrow \ #true\]
A museum offers free admission for people who arrive after 5 pm. Otherwise, the cost of admission is based on a person’s age: age 10 and under are charged $5 and everyone else is charged $10.

Exercise

Complete the function (admission after5? age) that returns the admission cost.

;; admission: Bool Nat -> Num
Sometimes it is desirable to flatten the conditionals.

\[
\text{define (admission after5? age)} \\
| \text{cond} | [after5? 0] \\
| \text{else} | (\text{cond} | [(<= age 10) 5] \text{else 10}] \\
| \text{else 10}] )
\]
Uses of cond

Conditionals can be used like any other expression:

```lisp
(define (add-1-if-even n)
  (+ n
     (cond
       [(even? n) 1]
       [else 0])))

(or (= x 0)
  (cond
    [(positive? x) (> x 100)]
    [else (< x -100)]))
```
Black-box and white-box testing

“In science, computing, and engineering, a **black box** is a device... which can be viewed in terms of its inputs and outputs, without any knowledge of its internal workings.” (Wikipedia)

Black-box testing refers to testing without reference to how the program works. Black-box tests should be written before you write your code. Your examples are black-box tests.

“A white box is a subsystem whose internals can be viewed but usually not altered.” (Wikipedia)

White-box testing should exercise every line of code. Design a test to check both sides of every question in every **cond**. These tests are designed after you write your code, by looking at how the code works.
Many languages have **static typing**. Here the type of every value and parameter is specified as part of the code. If you break the “contract” the code will not compile. Consider the following functions written in C:

```c
typedef struct {
    float x;
    float y;
} Posn;

float distance (Posn p1, Posn p2);

int main(void) {
    Posn p1 = {2.0, 5.0};
    Posn p2 = {3.0, 4.0};
    distance(p1, p2);
}
```

This works fine! But if I add `distance(p1, 3);` it gives this error:

```
distance.c: In function 'main':
distance.c:20:16: error: incompatible type for argument 2 of 'distance'
    distance(p1, 3);
```  

This can make it easier to write and debug your code.
Other languages, including Racket, have **dynamic typing**. This gives certain flexibility:

```scheme
;; (classify n) return "even" if n is even, "negative" if negative. 
;; Otherwise return n. 
;; classify: Int -> (anyof Str Nat)

(define (classify n)
  (cond [(even? n) "even"]
        [(negative? n) "negative"]
        [else n]))

(define (explain n)
  (cond [(string? (classify n)) "even-or-negative"]
        [else "positive-and-odd" ]))
```

Here a function can return or consume values of different types in different contexts.

This _can_ make it easier to write and debug your code.
Dealing with dynamic typing

While Racket does not enforce contracts, we will always assume that contracts are followed.

Never call a function with arguments that violate the contract and requirements. If you desire to use one of your own helper functions in a way that violates its contract, that likely means you should modify its contract!
Module summary

- Become comfortable using \texttt{cond} expressions, \texttt{and}, \texttt{or}, and \texttt{not}.
- Get used to combining these statements with the rest of our tools.
- Test these expressions, and know what black-box and white-box testing are.
- Make sure you understand short-circuiting in \texttt{and} and \texttt{or}.
- Become skillful at tracing code which includes \texttt{cond}, \texttt{and}, and \texttt{or}.

Before we begin the next module, please

- Read the Wikipedia entry on \textit{Filter (higher order function)}.
- Read \textit{How to Design Programs}, Section 9.