Module 4: Making Decisions

What does "<" mean?

Consider the expression "$x < 5$".

In math class, it tells us something about x.
We might combine the state "$x < 5$" with the statements "x is even" and "x is a perfect square" to conclude "x is 4".

In Racket, "<" means something different. A variable such as `x` already has a value.

What does "<" mean?

Suppose I define a variable:

```
(define x 10)
```

Now I create a Racket expression as close to "$x > 5$" as possible:

```
(> x 5)
```

This is *asking* "is this true?"
The statement "$x > 5$" can only be true or false. Which one?

If we evaluate `(> x 5)`, we substitute in the value of the constant, so our expression becomes `(> 10 5)`.
Since it is true that $10 > 5$, the statement evaluates to **`#true`**.

On the other hand, if I define the variable:

```
(define y 2)
```

Now `(> y 5)` => `(> 2 5)` => **`#false`** since it is not true that $2 > 5$.

Booleans (`Bool`)

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

$$\texttt{(< 4 6)} \longleftrightarrow 4 < 6$$
$$\texttt{(> 4 6)} \longleftrightarrow 4 > 6$$
$$\texttt{(= 5 7)} \longleftrightarrow 5 = 7$$
$$\texttt{(>= 5 5)} \longleftrightarrow 5 \geq 5$$
$$\texttt{(<= 5 5)} \longleftrightarrow 5 \leq 5$$

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

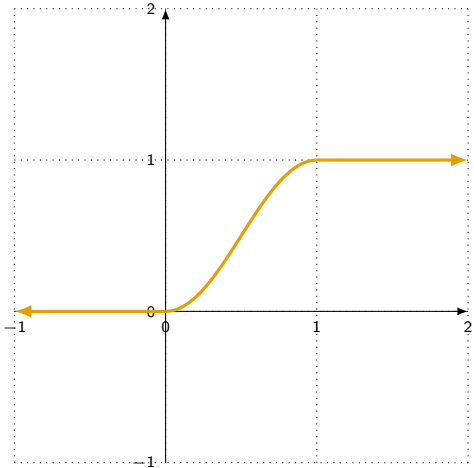
Predicates

A function which returns a `Bool` is called a **predicate**. For many predicates in Racket, the name ends with `?`.

Exercise

Figure out how to use each predicate in DrRacket.
Be sure you understand when each returns **`#true`** and when it returns **`#false`**.

1. `>`
2. **`even?`**
3. `string>=?`
4. `=`
5. **`equal?`**



A sin-squared window, used in signal processing, can be described by the following piecewise function:

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

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

Using cond to describe cases

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

```
(define (ssqw x)
  (cond
    [(< x 0) 0]
    [(>= x 1) 1]
    [(< 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.

```
(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)))])
```

Evaluate (< 4 0). This is **#false**. So look at the next clause.

Evaluate (>= 4 1). This is **#true**, so the expression evaluates to the answer, which is 1.

```
=> 1
```

Evaluating a cond statement

Exercise

Given the definition:

```
(define (foo x)
  (cond [(odd? x) "odd"]
        [(= 2 (remainder x 10)) "strange"]
        [(> x 100) "big"]
        [(even? x) "even"]))
```

First evaluate the expression by hand:

```
(foo 102)
```

Then run the code to check your answer.

Problem solving with cond

An example to remind us of the syntax of **cond**:

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

Exercise

Use **cond** to write a function `(absolute-value n)` which returns $|n|$.
(There is a built-in function **abs** which does this, but don't use it now.)

Consider that one way to define absolute value is as follows:

$$a(n) = \begin{cases} -n & \text{if } n < 0 \\ n & \text{if } n \geq 0 \end{cases}$$

No satisfied questions

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

```
(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.

else

We *could* use a question which always evaluates to `#true`:

It's always the case that $3 < 7$:

```
(define (absolute-value n)
  (cond
    [(> n 0) n]
    [(< 3 7) (- n)]))
```

Even simpler: `#true` is always `#true`:

```
(define (absolute-value n)
  (cond
    [(> n 0) n]
    [#true (- n)]))
```

Remember: the question/answer pairs are considered *in order*, top to bottom, and it stops as soon as it finds a question which evaluates to `#true`.
If no question evaluates to `#true` until it gets to `(< 3 7)`, that code will run.

else

This is useful sufficiently frequently that there is special keyword for it: **else**.

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

Exercise

Figure out what you think the following program will display.
Then run it in Racket to check your understanding.

```
(define (waldo x)
  (cond
   [(even? x) "even"]
   [#true "neither even nor odd"]
   [(odd? x) "odd"]
   ))

(waldo 4)
(waldo 3)
```

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

```
(cond
 [question1 answer1]
 [question2 answer2]
 ...
 [questionk answerk])
```

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

- `(cond [#false exp0][exp1 exp2]...)` => `(cond [exp1 exp2]...)`
- `(cond [#true exp0][exp1 exp2]...)` => `exp0`
- `(cond [else exp0])` => `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.
Verify your answer by comparing to the computer.

Testing cond

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 $x < 0$, 1 for integers $0 \leq x \leq 10$, 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.

```
;; (categorize n) return 0 for negative,
;;   1 for non-negative <= 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) 1)
```

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

(define (fix-list M)
  (map fix-limit M))
```

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

Exercise

Using **cond** and **map**, write a function `neg-odd` that consumes a `(listof Nat)`. The function returns a `(listof Int)` where all odd numbers are made negative, and all even numbers made positive.

```
(check-expect (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:

- `(and (> 5 4) (> 7 2)) => #true`
- `(or (>= 5 4) (> 7 2)) => #true`
- `(and (>= 5 5) (<= 7 2) (> 5 1)) => #false`
- `(or (> 4 5) (> 2 7) (< 9 4)) => #false`
- `(not (= 5 4)) => #true`
- `(and #true (< 3 7) (>= 9 1)) => #true`

Exercise

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

- `"big"` if $80 < x \leq 100$,
- `"small"` if $0 < x \leq 80$,
- `"invalid"` otherwise.

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

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:

```
(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 and

Use the following rules for tracing **and**:

- `(and #true exp ...)` => `(and exp ...)`
- `(and #false exp ...)` => `#false`
- `(and)` => `#true`

Exercise

Perform a trace of

```
(and (= 3 3) (> 7 4) (< 7 4) (> 0 (/ 3 0)))
```

Compare your trace to the solution on the next slide.

Substitution rules for and

```
=> (and #true (> 7 4) (< 7 4) (> 0 (/ 3 0)))
=> (and (> 7 4) (< 7 4) (> 0 (/ 3 0)))
=> (and #true (< 7 4) (> 0 (/ 3 0)))
=> (and (< 7 4) (> 0 (/ 3 0)))
=> (and #false (> 0 (/ 3 0)))
=> #false
```

Substitution rules for or

Use the following rules for tracing **or**:

- `(or #true exp ...)` => `#true`
- `(or #false exp ...)` => `(or exp ...)`
- `(or)` => `#false`

Exercise

Perform a trace of

```
(or (< 7 4) (= 3 3) (> 7 4) (> 0 (/ 3 0)))
```

Compare your trace to the solution on the next slide.

Substitution rules for or

```
=> (or #false (= 3 3) (> 7 4) (> 0 (/ 3 0)))
=> (or (= 3 3) (> 7 4) (> 0 (/ 3 0)))
=> (or #true (> 7 4) (> 0 (/ 3 0)))
=> #true
```

Nested Conditionals

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
```

Hint

`after5?` is a constant that is a `Bool`.

So it can be directly used as a question in a `[question answer]` pair in a **`cond`**.

```
(cond
[after5? ...]
```

Flattening Nested Conditionals

Sometimes it is desirable to flatten conditionals.
That is, instead of having a **cond** with another **cond** inside, we can rework them so they are multiple clauses of a single **cond**.
We can take the **not** of whatever is in the first **cond**, then **and** it with the inner **cond**.
We can often simplify even further. Here is an example:

```
; cond inside cond
(define (admission
    after5? age)
  (cond
    [after5? 0]
    [else
     (cond
       [(<= age 10) 5]
       [else 10]
       )
     ]))
```

↔

```
; "and" together 2 conds
(define (admission
    after5? age)
  (cond
    [after5? 0]
    [(and
      (not after5?)
      (<= age 10)) 5]
    [else 10]))
```

↔

```
; further simplified
(define (admission
          after5? age)
  (cond
    [after5? 0]
    [(<= age 10) 5]
    [else 10]))
```

Exercise

Flatten the `cond` expressions in the following function so there is only one `cond` with four `[question answer]` pairs.

```
;; (flatten-me x) Say which interval x is in.
;; flatten-me:  Nat -> Str

(define (flatten-me x)
  (cond [(< x 50)
         (cond [(< x 25) "first"]
               [else "second"])]
        [else
         (cond [(< x 75) "third"]
               [else "fourth"])]))
```

The character datatype `Char`

There is another datatype, `Char`, which represents a single character.
Some `Char` values include `#\a`, `#\4`, `#\space`, `#\!`

Here are a handful of the most useful functions to use with `Char`.

Check equality using **`char=?`** or **`equal?`**

```
(char=? #\e #\e) => #true
(char=? #\# #\&) => #false
(equal? #\X "X") => #false
(equal? #\7 7) => #false
```

To check if two `Char` values are in order:

```
(char<? #\e #\q) => #true
(char<? #\e #\Q) => #false
```

...and ignoring uppercase/lowercase:

```
(char-ci=? #\t #\T) => #true
(char-ci<? #\e #\Q) => #true
```

Documentation for `Char` is with that for `Str`, on the course website.

Strings as lists of characters

In several ways, a `Str` resembles a list.
Both have a length, and there are ways to get values from both:

```
(length (list 6 42 7)) => 3
(string-length "hello world!") => 12
(second (list 6 42 7)) => 42
(substring "hello world!" 1 2) => "e"
```

We can convert a `Str` to a `(listof Char)` and back, using **`string->list`** and **`list->string`**.

```
(string->list "hi there") => (list #\h #\i #\space #\t #\h #\e #\r #\e)
(list->string (list #\h #\i #\space #\t #\h #\e #\r #\e) => "hi there"
```

Exercise

Write a function `drop-e` that converts a `Str` to a `(listof Char)`, replaces each `#\e` with a `#\*`, and converts it back to a `Str`.

```
(check-expect (drop-e "hello world, how are you?")
              "h*llo world, how ar* you?")
```

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**.

Consider writing white box tests for this code:

```
(define (admission
         after5? age)
  (cond
    [after5? 0]
    [(<= age 10) 5]
    [else 10]))
```

For every `[question answer]` pair we need at least one test where the `question` is the first that evaluates to `#true` Here is one suggestion of tests:

1. `(check-expect (admission #true 42) 0)` to test the first `answer`.
2. `(check-expect (admission #false 7) 5)` to test the second `answer`.
3. `(check-expect (admission #false 42) 10)` to test the third `answer`.

Additional tests are desirable to check *edge cases*. These help us verify that we did what we meant to do: did we really mean to use `<=` instead of `<` ?
Testing with `age` of `9`, `10`, and `11` would cover the edge cases.

You can use the built-in *type predicates* to tell what type a value is:

```
(number? 42) => #true
(char? 42) => #false
(string? 42) => #false
(boolean? 42) => #false
```

Using this we can write a function that can consume values of different types. For example:

```
;; (any->string x) return a string representing x.
;; any->string: (anyof Str Num Bool) -> Str
;; Examples:
(check-expect (any->string "foo") "foo")
(check-expect (any->string 42) "42")
(check-expect (any->string #true) "#true")

(define (any->string x)
  (cond [(string? x) x]
        [(boolean? x)
         (cond [x "#true"]
               [else "#false"])]
        [(number? x) (number->string x)]))
```

Type Predicates and Contracts

Take a close look at the contact for this function:

```
;; (any->string x) return a string representing x.
;; any->string: (anyof Str Num Bool) -> Str
```

Where you see `(anyof ...)`, that represents a single parameter, that can have any of the type in the brackets.

The ability to have parameters with different types is called *dynamic typing*. Some languages instead have *static typing*, where each parameter can have only one type.

Static and dynamic typing have their advantages and disadvantages. But if we are using dynamic typing, it is important to use the contract to keep track of what the types are!

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!

If necessary, you may use `anyof` in your contract. But first consider if there is a way to accomplish you goal using a simpler contract. It is usually bad style to write functions that can consume multiple types (and a bad habit to have if someday you learn a statically-typed language).

Module summary

- Become comfortable using **`cond`** expressions, **`and`**, **`or`**, and **`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 **`and`** and **`or`**.
- Become skillful at tracing code which includes **`cond`**, **`and`**, and **`or`**.
- Be able to convert between a `Str` and a `(listof Char)`.

Further Reading: *How to Design Programs*, Section 4.

Before we begin the next module, please

- Read the Wikipedia entry on *Filter (higher order function)*.