Consider the expression “$x < 5$”.

In math class, it tells us something about $x$.

We might combine the statement “$x < 5$” with the statements “$x$ is even” and “$x$ is a perfect square” to conclude “$x$ is 4”.

In Racket, “<” means something different. A constant such as $x$ already has a value.
Suppose I define a constant:

```
(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 constant:

```
(define y 2)
```

Now $(> y 5) \Rightarrow (> 2 5) \Rightarrow false$ since it is not the case that $2 > 5$. 
Booleans (\texttt{Bool})

\(<, >, \leq, \geq\), and \(=\) are new functions, each of which produces a Boolean value (\texttt{Bool}).

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

Each produces \texttt{true} or \texttt{false}. These are the only values a \texttt{Bool} may take.
A function which produces a \texttt{Bool} is called a \textit{predicate}. For many predicates in Racket, the name ends with \texttt{?}.

**Figure out how to use each predicate in DrRacket.**
Be sure you understand when each produces \texttt{true} and when it produces \texttt{false}.

1 >
2 even?
3 =

The \texttt{member?} predicate consumes an \texttt{Any} and a \texttt{(listof Any)}.
It can be very handy. Make sure you understand its behaviour.
We also have several predicates that operate on strings.

We can tell if two strings are the same:

\[
\text{string=? "pearls" "gems"} \Rightarrow \text{false}
\]
\[
\text{string=? "pearls" "pearls"} \Rightarrow \text{true}
\]

We can also tell if a pair of strings are in alphabetic order. If one string comes before another, it is “less than” it. If it comes after, it is “greater than”. Some examples:

\[
\text{string<? "pearls" "swine"} \Rightarrow \text{true} \quad \text{; "pearls" before "swine".}
\]
\[
\text{string<? "pearls" "pasta"} \Rightarrow \text{false} \quad \text{; the "e" should come after the "a".}
\]
\[
\text{string>? "kneel" "zod"} \Rightarrow \text{false} \quad \text{; "kneel" before "zod".}
\]
\[
\text{string<=? "pearls" "pearls"} \Rightarrow \text{true}
\]
\[
\text{string=>? "pearls" "pearls"} \Rightarrow \text{true}
\]
A Boolean value by itself is not so useful. They become useful when they allow us to make decisions: to do different things in different situations.

Example: taking the absolute value of $x$.

$$|x| = \begin{cases} 
-x & \text{when } x < 0 \\
x & \text{when } x \geq 0 
\end{cases}$$
In Racket, we can compute $|x|$ with the **conditional expression**

```
(cond [(< x 0) (- x)]
     [(>= x 0) x])
```

- Conditional expressions use the special form `cond`.
- Each argument is a question/answer pair.
- The **question** is a Boolean expression.
- The **answer** is a possible value of the conditional expression.
- Square brackets are used by convention, for readability.
- Square brackets and parentheses are equivalent in the teaching languages (must be nested properly).
- `abs` is a built-in function in Racket.
A sin-squared window, used in signal processing, can be described by the following piecewise function:

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

Use \texttt{cond} to write a function \(\text{ssqw } x\) that behaves like \(f(x)\) above. Test it!
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` produces the answer.

For example, consider `(ssqw 4)`.

⇒ `(cond [(< 4 0) 0] [(>= 4 1) 1] [(< 4 1) (sqr (sin (* x 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`
Given the definition:

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

First evaluate the expression by hand:

```lisp
(foo 102)
```

Then run the code to check your answer.
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.
We *could* use a question which always evaluates to `true`:

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

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

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

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

\begin{verbatim}
(define (absolute-value n)
  (cond
    [(> n 0) n]
    [else (- n)]))
\end{verbatim}
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)
Recall we are imagining interpreting our programs as a series of substitutions, called a trace. Earlier we were a little vague. How exactly do we trace \texttt{cond}?

The general form of a conditional is

\begin{verbatim}
(cond
 [question1 answer1]
 [question2 answer2]
 ...
 [questionk answerk])
\end{verbatim}

To evaluate the conditional, evaluate \texttt{question1}, then perform the following substitutions:

1. \((\texttt{cond} \ [\texttt{false exp}]...) \Rightarrow (\texttt{cond} \ ...)
2. \((\texttt{cond} \ [\texttt{true exp}]...) \Rightarrow \texttt{exp}
3. \((\texttt{cond} \ [\texttt{else exp}]) \Rightarrow \texttt{exp}

Exercise
Perform a complete trace of this program.
Verify your answer by comparing to the computer.
Tests for conditional expressions

- Write at least one test for each possible answer in the expression.
- That test should be simple and direct, aimed at testing that answer.
- When the problem contains **boundary conditions** (like the cut-off between passing and failing), they should be tested explicitly.
- DrRacket highlights unused code.
For the example above:

```
(define (course-after-cs135 grade)
  (cond [(< grade 40) 'CS115]
        [(< grade 50) 'CS135]
        [(< grade 60) 'CS116]
        [else 'CS136]))
```

there are four intervals and three boundary points, so seven tests are required (for instance, 30, 40, 45, 50, 55, 60, 70).
Conditionals can be very useful in combination with map; it allows us to transform each item in a list, using a function. If that function includes cond statements, it allows us to do much more.

Here is an example of a function that does a simple operation to each item in a list, where the calculation uses cond.

```
;; (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
(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)
(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 produces 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))`
We combine predicates using the special forms \texttt{and}, \texttt{or}, and the function \texttt{not}. These all consume and produce \texttt{Bool} values.

- \texttt{and} produces \texttt{false} if at least one of its arguments is \texttt{false}, and \texttt{true} otherwise.
- \texttt{or} produces \texttt{true} if at least one of its arguments is \texttt{true} and \texttt{false} otherwise.
- \texttt{not} produces \texttt{true} if its argument is \texttt{false}, and \texttt{false} if its argument is \texttt{true}.

A few examples:

- \((\texttt{and} (\texttt{> 5 4}) (\texttt{> 7 2})) \Rightarrow \texttt{true}\)
- \((\texttt{or} (\texttt{>= 5 4}) (\texttt{> 7 2})) \Rightarrow \texttt{true}\)
- \((\texttt{and} (\texttt{> 5 5}) (\texttt{<= 7 2}) (\texttt{> 5 1})) \Rightarrow \texttt{false}\)
- \((\texttt{or} (\texttt{> 4 5}) (\texttt{> 2 7}) (\texttt{< 9 4})) \Rightarrow \texttt{false}\)
- \((\texttt{not} (= 5 4)) \Rightarrow \texttt{true}\)
- \((\texttt{and} \texttt{true} (\texttt{< 3 7}) (\texttt{>= 9 1})) \Rightarrow \texttt{true}\)

Both \texttt{or} and \texttt{and} require at least two arguments, but may have more.
Write a function that consumes an `Int`, and produces

- "baz" for even numbers in the interval [10, 40]
- "qux" for odd numbers in the interval [−20, 20]
- "xyzzy" for numbers less than −100 or greater than 200
- "corge" 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:

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

Attempting to evaluate `(/ 1 0)` would cause a division by zero error. So we might think we cannot call `(baz 0)`. But when `x` is zero, the first argument of **and** is *false*, so the second argument is not evaluated.
Substitution rules for \texttt{and}

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

- \((\texttt{and true expr ...}) \Rightarrow (\texttt{and expr ...})\)
- \((\texttt{and false expr ...}) \Rightarrow \texttt{false}\)
- \((\texttt{and}) \Rightarrow \texttt{true}\)

**Exercise**

Perform a trace of:
\[(\texttt{and (= 3 3) (> 7 4) (< 7 4) (> 0 (/ 3 0)))}\]

After you complete your trace, use the computer to verify your understanding.

**Exercise**

Perform a trace of:
\[(\texttt{define s "bravo"})
   (\texttt{and (> 7 4) true (string=? s "bravo"))}\]

After you complete your trace, use the computer to verify your understanding.
Substitution rules for **or**

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

- 
  - (or true expr ...) ⇒ true
- 
  - (or false expr ...) ⇒ (or expr ...)
- 
  - (or) ⇒ false

**Exercise**

Perform a trace of

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

After you complete your trace, use the computer to verify your understanding.

**Exercise**

Perform a trace of

(define s "bravo")
(or (< 7 4) false (string=? s "hooray"))

After you complete your trace, use the computer to verify your understanding.
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 \((\text{admission after5? age})\) that produces the admission cost.

;; admission: Bool Nat -> Num

Hint

\(\text{after5?}\) is a constant that is a \text{Bool}.

So it can be directly used as a question in a \([\text{question answer}]\) pair in a \text{cond}.

\((\text{cond}

[\text{after5?} \ldots])\)
Often “flat” conditionals are easier to read than “nested” conditionals.

That is, instead of having a \texttt{cond} with another \texttt{cond} inside, we can rework them so they are multiple clauses of a single \texttt{cond}.

We can take the \texttt{not} of whatever is in the first \texttt{cond}, then \texttt{and} it with the inner \texttt{cond}.

We can often simplify even further. Here is an example:

\begin{verbatim}
; cond inside cond
(define (admission after5? age)
  (cond
    [after5? 0]
    [else
      (cond
        [(<= age 10) 5]
        [else 10]]))))

\end{verbatim}

\begin{verbatim}
; "and" together 2 conds
(define (admission after5? age)
  (cond
    [after5? 0]
    [(and
      (not after5?)
      [(<= age 10) 5]
      [else 10]]))

\end{verbatim}

\begin{verbatim}
; further simplified
(define (admission after5? age)
  (cond
    [after5? 0]
    [(<= age 10) 5]
    [else 10]))
\end{verbatim}
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`

```scheme
(define (flatten-me x)
  (cond [(< x 50)
          (cond [(< x 25) "first"
                 [else "second"]])
          [else
           (cond [(< x 75) "third"
                  [else "fourth"])])))
```
The character datatype \texttt{Char}

There is another datatype, \texttt{Char}, which represents a single character.

Some \texttt{Char} values include \texttt{\#a}, \texttt{\#4}, \texttt{\#space}, \texttt{\#!}

Here are a handful of the most useful functions to use with \texttt{Char}.

Check equality:

\begin{align*}
(\texttt{char}=? \texttt{\#\e} \texttt{\#\e}) & \Rightarrow \texttt{true} \\
(\texttt{char=? \#\# \#\&}) & \Rightarrow \texttt{false}
\end{align*}

Check if \texttt{Char} values are in alphabetic order:

\begin{align*}
(\texttt{char<? \#\e \#q}) & \Rightarrow \texttt{true} \\
(\texttt{char<? \#Q \#e}) & \Rightarrow \texttt{true} \\
(\texttt{char<=? \#e \#e}) & \Rightarrow \texttt{true} \\
(\texttt{char>=? \#e \#f}) & \Rightarrow \texttt{false}
\end{align*}

Check if a value is a \texttt{Char}:

\begin{align*}
(\texttt{char? \#!}) & \Rightarrow \texttt{true} \\
(\texttt{char? "!"}) & \Rightarrow \texttt{false}
\end{align*}
Write a function that consumes a value of any type, and determines if the value is the Char value #\X.

;; (is-char-X? v) produce true if v is #\X, false otherwise.
(chk-expect (is-char-X? (list 3 2 1 "blastoff")) false)
(chk-expect (is-char-X? 9) false)
(chk-expect (is-char-X? "X") false)
(chk-expect (is-char-X? #\X) true)

;; is-char-X?: Any -> Bool
In several ways, a \texttt{Str} resembles a list.

Both have a length, and there are ways to get values from both:

\[
\begin{align*}
\text{(length (list 6 42 7))} & \Rightarrow 3 \\
\text{(string-length "hello world!")} & \Rightarrow 12 \\
\text{(first (list 6 42 7))} & \Rightarrow 6 \\
\text{(substring "hello world!" 0 1)} & \Rightarrow "h"
\end{align*}
\]

We can convert a \texttt{Str} to a (listof \texttt{Char}) and back, using \texttt{string->list} and \texttt{list->string}.

\[
\begin{align*}
\text{(string->list "hi there")} & \Rightarrow (\text{list #\h #\i \space #\t #\h #\e #\r #\e}) \\
\text{(list->string (list #\h #\i \space #\t #\h #\e #\r #\e))} & \Rightarrow "hi there"
\end{align*}
\]

Since we have tools to work with lists, this lets us do many things with \texttt{Str}.

\begin{itemize}
\item \textbf{Exercise} Write a function \texttt{drop-e} that converts a \texttt{Str} to a (listof \texttt{Char}), replaces each #\e with #\*, and converts it back to a \texttt{Str}.
\[
\begin{align*}
\text{(check-expect (drop-e "hello world, how are you?")}
\end{align*}
\]
\[
\begin{align*}
"h*llo world, how ar* you?"
\end{align*}
\]
\end{itemize}
Some of your tests, including your examples, will have been defined before the body of the function was written.

These are known as closed-box tests, because they are not based on details of the code.

Other tests may depend on the code, for example, to check specific answers in conditional expressions.

These are known as open-box tests. Both types of tests are important.
You can use the built-in *type predicates* to tell what type a value is:

\[
\begin{align*}
  \text{(number? 42)} & \Rightarrow \text{true} \\
  \text{(string? 42)} & \Rightarrow \text{false} \\
  \text{(integer? 3.14)} & \Rightarrow \text{false} \\
  \text{(char? 42)} & \Rightarrow \text{false} \\
  \text{(boolean? 42)} & \Rightarrow \text{false} \\
  \text{(list? (list 2 3))} & \Rightarrow \text{true}
\end{align*}
\]
Using type predicates 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)])

In general, though, it’s better to avoid consuming different types.
Take a close look at the contact for this function:

```haskell
;; (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 since 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 \texttt{anyof} in your contract. But first consider if there is a way to accomplish your 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).
I wish to develop a predicate `cat-start-or-end?`, which consumes a `Str` and determines if the `Str` starts or ends with "cat".

What might we need to know?

- Is it long enough?
- Does it start with "cat"?
- Does it end with "cat"?
Write a function `cat-start-or-end?` which consumes a `Str` and determines if it starts or ends with "cat".
In the land of Yendor, the currency is the zorkmid, zm.

Taxes are calculated as follows:

- For incomes under 45,000 zm, 15% is paid as tax.
- For incomes between 45,000 zm and 90,000 zm, 20% of each additional zm is paid as tax.
- For incomes above 90,000 zm, 25% of each additional zm is paid as tax.

**Exercise**

Define appropriate constants, then create a function `tax-payable` that consumes a `Num` representing income, and produces the taxes to be paid on that income.
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).
Summary: built-in functions

In this module we added the following to our toolbox:

  integer? list->string list? member? not number? odd? or string->list string<=? string<?
  string=? string>=? string>? string?

These are the functions and special forms currently in our toolbox:

  check-expect check-within cond define else even? exp expt first floor foldl foldr
  integer? length list list->string list? map max member? min not number->string number?
  odd? or quotient range remainder rest sqr sqrt string->list string-append string-length