What do you suppose this code does?

```scheme
(define (d-over n d)
  (cond
    [(< n (sqr d)) true]
    [(= 0 (remainder n d)) false]
    [else (d-over n (+ 1 d))]))

(define (p? n)
  (and (not (= n 1)) (d-over n 2)))
```
;; (d-over n d) produce false if d or larger divides n; else true.
;; d-over: Nat Nat -> Bool
(define (d-over n d)
  (cond
    [(< n (sqr d)) true]
    [ (= 0 (remainder n d)) false]
    [else (d-over n (+ 1 d))]))

;; (prime? n) produce true if n is prime; false otherwise.
;; Examples:
(check-expect (prime? 9) false)
(check-expect (prime? 17) true)

;; prime?: Nat -> Bool
(define (prime? n)
  (and (not (= n 1)) (d-over n 2)))
Every program is an act of communication:

- With the computer
- With yourself in the future
- With other programmers

By writing code, we communicate with the computer by saying exactly what it shall do.

For other programmers we want to communicate other things, such as:

- What are we trying to do?
- Why are we doing it this way?

We communicate with other programmers (and ourselves) using comments.
Comments let us write notes to ourselves or other programmers.

;; By convention, please use two semicolons, like
;; this, for comments which use a whole line.

(+ 6 7) ; comments after code use one semicolon.

;; Let's define some constants:
(define year-days 365) ; not a leap year

In DrRacket there is a command Racket → Comment Out with a Box. **Never** use this command! It makes your assignment impossible to mark.
The Design Recipe is a set of comments and special code that we write for each function.

The design recipe has three main purposes:

1. Writing it helps you understand the problem. If you cannot write the design recipe, you probably don’t actually understand the problem.

2. It helps you write understandable code so you or another programmer can read it.

3. It helps you write tested code so you have some confidence it does what is should.

You should use the design recipe for every function you write.
See the definition of \((e^{10} \ n)\). Carefully read the design recipe.

**design recipe**

\[
\begin{align*}
&\text{;; (e^{10} \ n) produce 1 followed by n zeros.} \\
&\text{;; Examples:} \\
&\text{(check-expect (e^{10} \ 2) 100)} \\
&\text{(check-expect (e^{10} \ 5) 100000)} \\
&\text{(check-expect (e^{10} \ 0) 1)} \\
&\text{;; e^{10}: Nat -> Nat} \\
\end{align*}
\]

**implementation**

\[
\begin{align*}
&(\text{define (e^{10} \ n)} \\
&\qquad ((\text{lambda (+ -)}\quad (- + -) \quad ; \text{ <-- Tie fighter!}) \\
&\qquad\quad ) \ n \ (\text{lambda (+ /)}) \\
&\qquad\quad (\text{cond} \ [ (= \ + \ 0) \ 1] \\
&\qquad\qquad\quad [(= \ + \ +) \ (* \ 10 \ (/ \ (- \ + \ 1) \ /))])\))\))
\end{align*}
\]

(The implementation is correct, but will not work in the *Beginning Student* language. It is intentionally hard to read. **Read the design recipe only!**)
What does check-expect mean?

These lines say something weird:

(check-expect (e10 2) 100)
(check-expect (e10 5) 100000)
(check-expect (e10 0) 1)

(check-expect expression expected-expression) is a special form that we use for examples and tests.

When we write a check-expect we fill in as follows:

- expression is the function call we are testing.
- expected-expression is the “correct answer” that we calculated.

So here we are saying that if the e10 function is properly written, it should be that \((\text{e10} 2)\) produces 100, \((\text{e10} 5)\) produces 100000, and \((\text{e10} 0)\) produces 1.

This both helps us understand the function, and demonstrate that our code works properly.
1. The **purpose** describes what the function calculates. Explain the role of every parameter.
   
   ```scheme```
   ;; (prime? n) produce true if n is prime; false otherwise.
   ```

2. Choose **examples** which help the reader understand the purpose.
   ```scheme```
   (check-expect (prime? 9) false)
   (check-expect (prime? 17) true)
   ```

3. The **contract** indicates the type of arguments the function consumes and the value it produces. Can be **Num**, **Int**, **Nat**, or other types. Types are always Capitalized.
   
   ```scheme```
   ;; prime?: Nat -> Bool
   ```

4. The **implementation** is interpreted by the computer.
   ```scheme```
   (define (prime? n)
     (and (not (= n 1)) (d-over n 2)))
   ```

5. The **tests** resemble examples, but are chosen to try to find bugs in the implementation.
   ```scheme```
   ;; Tests
   (check-expect (prime? 1) false)
   (check-expect (prime? 982451653) true)
   ```
Here is an example of the design recipe for a function `distance` which computes the distance between \((0, 0)\) and a given point \((x, y)\). (That is, \(\text{distance}(x, y) = \sqrt{x^2 + y^2}\).)

**purpose**

```scheme
;; (distance x y) produce the distance between (x,y)
;; and the origin.
```

**examples**

```scheme
;; Examples:
(check-expect (distance 7 0) 7)
(check-expect (distance 3 4) 5)
```

**contract**

```scheme
;; distance: Num Num -> Num
```

**implementation**

```scheme
(define (distance x y)
  (sqrt (+ (sqr x) (sqr y))))
```

**tests**

```scheme
;; Tests for distance:
(check-expect (distance -3 -4) 5)
(check-expect (distance 0 0) 0)
```
Write purpose, examples, contract, and tests for:

1. The absolute value function \( \text{abs} \ x \)

2. A function \( \text{gcd} \ a \ b \) which computes the GCD of two natural numbers

(Don’t write the implementations, we don’t have the tools yet!)
- Write the **purpose**, **examples**, and **contract** before the implementation!

- Use meaningful names for parameters and functions. 
  Don’t call a function **function**, or **test**. The name should suggest the purpose.

- Do not put types of parameters in the **purpose**; the **contract** contains this information.

- Use the most specific data type possible.
  For a number which could be any real value, use **Num**. If you know it’s an integer, use **Int**; if you further know it’s a natural number (an **Int ≥ 0**), use **Nat**. More types later.

- Write the **purpose**, **examples**, and **contract** before the implementation!
For **examples**, choose common usage. The point is to clarify what the function does.

Format for examples is `(check-expect function-call correct-answer)`. An example:

```scheme
(check-expect (gcd 40 25) 5)
```

Design **tests** to test different situations which may be tricky.

```scheme
(check-expect (gcd 42 0) 42)
(check-expect (prime? 1) false)
```

Any time there is a decision, such as even/odd, make sure you test all possibilities. More is not necessarily better!

Write the **purpose**, **examples**, and **contract** before the implementation!
Sometimes we write functions where certain inputs are not valid.

Consider this example. For this function, what inputs are invalid?

;;; (sqrt-shift x c) produces the square root of (x - c).
(check-expect (sqrt-shift 7 3) 2)
(check-within (sqrt-shift 9 7) 1.4142 0.0001)

;;; sqrt-shift: Num Num -> Num
;;; Requires: x - c >= 0
(define (sqrt-shift x c)
  (sqrt (- x c)))
We want to use numbers which are real, not complex, so we can’t take the square root of a negative number. So we need \( x - c \geq 0 \), equivalent to \( x \geq c \).

Add this as a Requires section:

\[
\text{;; (sqrt-shift x c) produces the square root of (x - c).}
\text{;; Examples:}
\text{(check-expect (sqrt-shift 7 3) 2)}
\text{(check-within (sqrt-shift 9 7) 1.4142 0.0001)}
\]

\[
\text{;; sqrt-shift: Num Num -> Num}
\text{;; Requires: x - c \geq 0}
\text{(define (sqrt-shift x c)}
  \text{(sqrt (- x c)))}
\]
Tests using `check-expect`

You have probably noticed that our tests look like

```
(check-expect (...) ...)
```

What is going on? Answer: `check-expect` is a built-in tool that we use for testing.

It has two parameters:

- The first is a call to the function being tested. This becomes the answer the function actually produces.
- The second is the correct answer, calculated by hand.

If the answer the function actually produces is the same as the correct answer, the function passes the test.

Never use your function to determine the value for the second parameter! If you do you are only demonstrating that your function does what it does.
Some functions produce *inexact* answers. Often this means it is an irrational number. For example:

\[
\text{(sqrt 2)} \Rightarrow \#i1.4142135623730951
\]

In this case, \(\text{(check-expect test-expression true-value)}\) will not work. (Try it.) Instead use \(\text{(check-within test-expression true-value max-error)}\)

For example,

\[
\text{(check-within (sqrt 2) 1.4142 0.0001)}
\]

**Write one more exact and one more inexact test for \textbf{sqrt-shift}**.

The course notes may omit portions of the style guide.

Exercise

Download and consider the style guide. Identify and carefully read important sections as the course progresses.

Consult the style guide on the course website for correct design recipe use. Your assignments will be graded on correct use of the design recipe!
A string is a value made up of letters, numbers, blanks, punctuation marks, and other characters, all enclosed in quotation marks.

Examples: "hat", "This is a string.", and "Module 2".

Strings are used extensively in programming. Anywhere you see text, there are strings operating behind the scenes.
Some Useful String Functions

Using string functions we do many things:

...stick two or more strings together:
\[
\text{(string-append } \text{"now" \text{"here"}) } \Rightarrow \text{"nowhere"}
\]
\[
\text{(string-append } \text{"he" \text{"llo " \text{"how" \text{" R" \text{"U?"}) } \Rightarrow \text{"hello how RU?"}
\]

...determine how many characters are in a string:
\[
\text{(string-length } \text{"length") } \Rightarrow 6
\]

...chop up a string:
\[
\text{(substring } \text{"caterpillar" 5 9) } \Rightarrow \text{"pill"
\]
\[
\text{(substring } \text{"cat" 0 0) } \Rightarrow \text{"
\]
\[
\text{(substring } \text{"nowhere" 3) } \Rightarrow \text{"here" } \text{; this goes to the end of the Str}
\]

...convert a number to a string:
\[
\text{(number->string } 42) \Rightarrow \text{"42"}
Use `string-append` and `substring` to complete the function `chop-word`:

```scheme
;; (chop-word s): select some pieces of s.
;; Examples:
(check-expect (chop-word "In a hole in the ground there lived a hobbit.")
  ;; ^ ^ ^ ^ ^ ^ ^ ^ ^
  ;; index: 0 5 10 15 20 25 30 35 40
  "a hobbit lived in the ground")
(check-expect (chop-word "In a town by the forest there lived a rabbit.")
  ;; ^ ^ ^ ^ ^ ^ ^ ^ ^
  ;; index: 0 5 10 15 20 25 30 35 40
  "a rabbit lived by the forest")
(check-expect (chop-word "ab c defg hi jkl mnopqr stuvw xyzAB C DEFGHIJ")
  "C DEFGHI xyzAB hi jkl mnopqr")
```

`chop-word`: Str -> Str
Use the constants `the-str` and `len-str`, along with the string functions `string-append`, `string-length`, and `number->string` to complete the function `describe-string`:

```
(define the-str "The string ")
(define len-str " has length ")

;; (describe-string s) Say a few words about s.
;; Examples:
(check-expect (describe-string "foo") "The string 'foo' has length 3")
(check-expect (describe-string ") "The string '' has length 0")

;; describe-string: Str -> Str
```
We are going to write a function `swap-parts` which consumes a `Str`, and produces a new `Str` which has the front and back halves reversed.

**Exercise**

Write the **purpose** for `swap-parts`.

**Exercise**

Write at least two **examples** for `swap-parts`.

**Exercise**

Write the **contract** for `swap-parts`. 
You should now have a clear purpose, examples, and contract for `swap-parts`.

So now we understand the problem, and any future programmers will understand what we are trying to accomplish.

Now it comes to write the implementation, the part the computer reads.
Thinking through the problem, it needs to be something like this:

\[
\text{(define (swap-parts s)}
\text{\hspace{1cm} (string-append)}
\text{\hspace{1cm} \ldots \; ; \text{an expression that is the back half of } s}
\text{\hspace{1cm} \ldots \; ; \text{an expression that is the front half of } s}
\text{\hspace{1cm})}
\]

Consider: now we have identified a distinct smaller problem, easier than the whole problem:

“find the back half of \( s \).”

Since this is a well defined problem, we can write another function to accomplish this task. Since it helps us solve our main function, we will call this a “helper function”.
A **helper function** is a function used by another function to

- generalize similar expressions
- express repeated computations
- perform smaller tasks required by your code
- improve readability of your code

Use meaningful names for all functions and parameters. Name should suggest purposes.

![Warning: Never call a helper function “helper”! Use meaningful names!](image)

Put helper functions below the functions they help.

See the Style Guide for further details.
Write the **purpose**, **contract**, and **examples**, for **back-part**.

Write the **implementation** for **back-part**.
Click “Run”. Make sure it passes the tests.
If it doesn’t pass, **debug now**, before your program gets too complicated.

Write the **purpose**, **contract**, and **examples** for **front-part**.

Write the **implementation** for **front-part**.
Again, click “Run”, and make sure it passes the tests. Run frequently; make sure your changes do what you expect.
Probably you have written something similar to this:

```lisp
;; (back-part s) return the back part of s.
;; back-part: Str -> Str
(define (back-part s)
  (substring s (quotient (string-length s) 2)))

;; (front-part s) return the front part of s.
;; front-part: Str -> Str
(define (front-part s)
  (substring s 0 (quotient (string-length s) 2)))
```

Notice that identical code appears twice: `(quotient (string-length s) 2)`. This snippet finds the location of the middle of `s`.

...That’s a distinct smaller problem, easier than the whole problem:

“Find the location of the middle of `s`.”

So we should write another helper function.
When we identify a problem that is smaller than the whole problem, it is often a good idea to write a helper to solve the smaller problem.

Since we have more that one copy of the same code, we should write a helper that does this calculation, and use it repeatedly.

Here the smaller problem is

“Find the location of the middle of $s$.”

We will create a function $\text{mid}$ to do this.

\begin{itemize}
  \item Write $\text{mid}$. Remember to follow the design recipe (purpose, examples, contract).
  \item Update $\text{back-part}$ and $\text{front-part}$ to use $\text{mid}$.
\end{itemize}
Ideally, the Design Recipe:

- provides a starting point for solving the problem.
- helps you understand the problem better.
- helps you write correct, reliable code.
- improves readability of your code.
- prevents you from losing marks on assignments!
Design Recipes are Required in Industry!

Students sometimes consider the design recipe as an afterthought, as “something annoying they make you do in school”. It’s not.

Take a look at the Google C++ Style Guide.

Write the purpose, examples, and contract before the implementation!
Know how to use the whole design recipe, and use it for all functions.

Get in the habit of writing your implementation last! Start with the design recipe.

Use `check-expect` and `check-within` to test your code.

Write helper functions when appropriate, again using the design recipe.

Work with `Str`, `Nat`, `Int`, and `Num`.

Before we begin the next module, please

- Read the Wikipedia entry on *Higher-order functions*.
- Read the Style Guide, Sections 1-4, 5.1-5.3 and 6.
In this module we added the following to our toolbox:

check-expect check-within number->string string-append string-length substring

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

* + - / abs ceiling check-expect check-within define exp expt floor max min number->string quotient remainder sqr sqrt string-append string-length substring