Module 2: The Design Recipe

In Racket, select Language → Choose language → Intermediate student.

If you have not already, make sure you

- Read *How to Design Programs*, Sections 1, 2, 3.
- Read the Survival Guide on assignment style and submission
- Read the Style Guide, Sections 1-4 and 6.
What do you suppose this code does?

```
(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)))
```

It’s quite difficult to figure out what this does, even if you understood all the pieces.
(define (d-over n d)
  (cond
   [(< n (sqr d)) #true]
   [ (= 0 (remainder n d)) #false]
   [else (d-over n (+ 1 d))])))

(define (prime? n)
  (and (not (= n 1)) (d-over n 2)))

It’s much easier to figure out what this program is supposed to do, even without understanding the code. The programs state their purpose!
Every program is an act of communication:

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

You **must not** use DrRacket’s comment boxes!
If you do, our submission system will not be able to read your assignment, so it will not be marked! (Basic tests may help you notice this.)
The Design Recipe

Two main purposes:

1. *Understandable code* so you or another programmer can read it
2. *Tested code* so you have some confidence it does what is should

Use the design recipe for every function you write.
Students sometimes consider the design recipe as an afterthought, as \textit{something annoying they make you do in school}. It’s not.

Take a look at the \textit{Google C++ Style Guide}.

In comparison, the CS115 style guide is quite short.
The Design Recipe

1. The **purpose** describes what the function calculates. Explain the role of every parameter.

   ```scheme
   ;; (prime? n) return #true if n is prime; #false otherwise.
   ``

2. The **contract** indicates the type of arguments the function consumes and the value it returns. Can be **Num**, **Int**, **Nat**, or other types.

   ```scheme
   ;; prime?: Nat -> Bool
   ``

3. Choose **examples** which help the reader understand the purpose.

   ```scheme
   ;; Examples:
   (check-expect (prime? 9) #false)
   (check-expect (prime? 17) #true)
   ``

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)
   ```
Best Practices for the Design Recipe

- Write the implementation after everything else. Seriously!
- Use meaningful names for parameters.
- 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 \geq 0`), use `Nat`. More types later.
- For **examples**, choose values which show common usage. The point is to clarify what the function does.
- Write examples before you write your code!
- Format for examples is (check-expect function-call correct-answer)
  (check-expect (gcd 40 25) 5)
- Design **tests** to test different situations which may be tricky:
  (check-expect (gcd 42 0) 42)
  (check-expect (prime? 1) #false)
Exercise
Write a full design recipe for a function distance which computes the distance between (0, 0) and a given point (x, y).
Include purpose, contract, examples, implementation, and tests.

;; (distance x y) return the distance between (x,y) and the origin.
;; Num Num -> Num
;; Examples:
(check-expect (distance 7 0) 7)
(check-expect (distance 3 4) 5)

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

;; Tests for distance:
(check-expect (distance -3 -4) 5)
(check-expect (distance 0 0) 0)
Consider the function

;;; (sqrt-shift x c) returns the square root of (x - c).
;;; sqrt-shift: Num Num -> Num
(define (sqrt-shift x c)
  (sqrt (- x c)))

What inputs are invalid?

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 \texttt{Requires} section:

;;; (sqrt-shift x c) returns the square root of (x - c).
;;; sqrt-shift: Num Num -> Num
;;; Requires: x \geq c
(define (sqrt-shift x c)
  (sqrt (- x c)))
Inexact tests

Some functions return *inexact* answers, for example,

```
(sqrt 2) ⇒ #i1.4142135623730951
```

In this case, (check-expect test-expression true-value) will not work. Use

```
(check-within test-expression true-value max-error)
```

```
(check-within (sqrt 2) 1.4142 0.0001)
```

Note this is necessary *only* if the answer is inexact, labelled with the #i prefix, or if your function is computing an approximation.

```
(check-expect (/ 1 2) 0.5) is fine.
```
“Return” or “produce”?

The textbook, style guide, and some other old books use the word “produce” to indicate the value created by a function. The modern approach is to use the word “return”.

In this course, you may use either term. But if you ever work in another language, you will almost certainly use “return”.

Consider the following function:

```
(define (f a b) (+ a b))
```

In the purpose of the function I could write either:

```
;; (f a b) return the sum of a and b.
```

or

```
;; (f a b) produce the sum of a and b.
```
A final note on the design recipe

The course notes may omit portions of the style guide.
Consult the style guide on the course website for correct design recipe use.
A **Str** is a value made up of letters, numbers, blanks, and punctuation marks, all enclosed in quotation marks. Examples: "hat", "This is a string.", and "Module 2".

### String functions

```scheme
(string-append "now" "here") => "nowhere"
(string-length "length") => 6

(substring "caterpillar" 5 9) => "pill"
(substring "cat" 0 0) => ""
(substring "nowhere" 3) => "here" ; go to the end of the Str

(string<? "pearsl" "swine") => #true ; "p" comes before "s".
(string<? "pearsl" "pasta") => #false ; first chars equal -- check next.
(string=? "pearsl" "gems") => #false

(number->string 42) => "42"
```

For more detail, see the documentation in DrRacket.
We are going to write a function `swap-parts` which consumes a `Str`, and returns a new `Str` which has the front and back halves reversed. If the length is odd, include the middle character with the second half.

Ex. Write the **purpose** and **contract** for `swap-parts`.

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

Ex. Write the **implementation** for `swap-parts`. ... We should use 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. (Never call one “helper”!) Put helper functions above any functions they help. See the style guide for further details.
Designing a Program

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

Ex. Write the **implementation** for front-part.

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

Ex. Write the **implementation** for back-part.

... We have duplicated code! We should use another helper function.

Ex. Write `mid` – remember, follow the design recipe.

Ex. Update front-part and back-part to use `mid`. 
Computing a phone bill

cell-bill consumes the number of daytime and evening minutes used and returns the total charge for minutes used.

Details of the plan:

- 100 free daytime minutes
- 200 free evening minutes
- $1 per minute for each additional daytime minute
- $0.5 per minute for each additional evening minute
(define free-day-m 100) ; how many free daytime minutes
(define free-eve-m 200) ; how many free evening minutes
(define cost-day-m 1) ; cost per day minute after limit
(define cost-eve-m 0.5) ; cost per evening minute after limit

;; (cell-bill dm em) return the cost of using a cell phone dm minutes in the
;; day and em in the evening.
;; cell-bill: Nat Nat -> Num
;; Example:
(check-expect (cell-bill 105 203) 6.5)

(define (cell-bill dm em)
  (+ (* (max 0 (- dm free-day-m)) cost-day-m)
     (* (max 0 (- em free-eve-m)) cost-eve-m)))

;; Tests:
(check-expect (cell-bill 100 200) 0)
;; probably quite a few more...
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!
Module summary

- 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.