CS115 - Module 2 - The Design Recipe

Cameron Morland

Fall 2017
(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!
;;; (d-over n d) produce false iff d or larger divides n.
;;; (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))])
)

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

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

It’s much easier to figure out what this program is supposed to do.
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, your assignments will not be marked!
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 *something annoying they make you do in school*. It's not. Witness the Google C++ Style Guide.
The Design Recipe

1. The **purpose** describes what the function calculates. Explain the role of every parameter.
   
   ;; (p? n) produce true if n is prime; false otherwise.

2. The **contract** indicates the type of arguments the function consumes and the value it produces. Can be `Num`, `Int`, `Nat`, or other types.
   
   ;; p?: Nat -> Bool

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

   ;; Examples:
   
   (check-expect (p? 9) false)
   (check-expect (p? 17) true)

4. The **implementation** is interpreted by the computer.

   (define (p? 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.

   ;; Tests
   
   (check-expect (p? 1) false)
   (check-expect (p? 982451653) true)
Best Practices for the Design Recipe

- Write the implementation after everything but tests. 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 ≥ 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)
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) produce 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) produces 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 Requires section:

;;; (sqrt-shift x c) produces the square root of (x - c).
;;; sqrt-shift: Num Num -> Num
;;; Requires: x >= c
(define (sqrt-shift x c)
  (sqrt (- x c)))
Some functions return *inexact* answers, for example,

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

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

\[(\text{check-expect test-expression true-value range})\]

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

Note this is necessary *only* if the answer is inexact, labelled with the \#i prefix.

\[(\text{check-expect } (/ 1 2) 0.5)\text{ is fine.}\]
A final note on the design recipe

The course notes, both these and the official ones, 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:
(string-append "now" "here") ⇒ "nowhere"
(string-length "length") ⇒ 6
(substring "caterpillar" 5 9) ⇒ "pill"
(substring "cat" 0 0) ⇒ ""
(substring "nowhere" 3) ⇒ "here"
We are going to write a function `swap-parts` which consumes a string, and produces a new string which has the front and back halves reversed. If the length is odd, include the middle character with the second half.

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

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

Write the body 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

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

Write the **body** for **front-part**.

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

Write the **body** for **back-part**.

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

Write **mid** – remember, follow the design recipe.

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 produces 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 daytime minute
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!
Know how to use the whole design recipe, and use it for all functions.
Write your implementation last!
Use check-expect and check-within to test your code.
Write helper functions when appropriate, again using the design recipe.
Work with strings.
See the official course notes for more details.

Before we begin the next module, please

- Read How to Design Programs, sections 4-5