Module 2: The Design Recipe

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

If you have not already, make sure you

- Read the Survival Guide on assignment style and submission.
- Read the Style Guide, Sections 1-4 and 6.

Exercise

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.

Communication

```
;; (d-over n d) return #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) return #true if n is prime; #false otherwise.
;; prime?: Nat -> Bool
;; Examples:
(check-expect (prime? 9) #false)
(check-expect (prime? 17) #true)

(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*!

Communication

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

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

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.

Exercise

See the definition of `(e10 n)`. Carefully read the design recipe.
What should be returned by the following expression? `(+ (* 4 (e10 3)) (* 2 (e10 2)))`

design recipe

```
;; (e10 n) return 1 followed by n zeros.
;; e10: Nat -> Nat
;; Examples:
(check-expect (e10 2) 100)
(check-expect (e10 5) 100000)
(check-expect (e10 0) 1)
```

implementation

```
(define (e10 n)
  ((lambda (+ -)      (- + -)      ; <-- Tie fighter!
    ) n (lambda (+ /)
          (cond [(= + 0) 1]
                [else (* 10 (/ (- + 1) /))]))))
```

(The implementation is correct, but will not work in the *Intermediate 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 function 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 `d10` function is properly written, it should be that `(e10 5)` becomes `100000`.

This both helps us understand the function, and demonstrate that our code works properly.

Defining the Design Recipe

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

```
;; (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. Types are always **C**apitalized.

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

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

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

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

```
(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.

```
;; Tests
(check-expect (prime? 1) #false)
(check-expect (prime? 982451653) #true)
```

Full Design Recipe Example

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, distance$(x, y) = \sqrt{x^2 + y^2}$.)

purpose
```
;; (distance x y) return the distance between (x,y)
;;   and the origin.
```

contract
```
;; distance: Num Num -> Num
```

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

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

tests
```
;; Tests for distance:
(check-expect (distance -3 -4) 5)
(check-expect (distance 0 0) 0)
```

Exercise

Write purpose, contract, examples, and tests for:

1. The absolute value function `(abs x)`
2. A function `(gcd a b)` which computes the GCD of two natural numbers

(Don't write the implementations, we don't have the tools yet!)

Best Practices for the Design Recipe

- Write the **purpose**, **contract**, and some **examples** 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**, **contract**, and some **examples** before the implementation!

Best Practices for the Design Recipe

- 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:
  ```
  (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 the **purpose**, **contract**, and some **examples** before the implementation!

Additional contract requirements

Sometimes we write functions where certain inputs are not valid.
There are many reasons this may happen. A few examples: a function might consume a `Num` that is a grade, and so a number between 0 and 100; another function might consider two `Nat` that must be relatively prime.
We will see further examples throughout the term.

Exercise

Consider the function:

```
;; (sqrt-shift x c) returns the square root of (x - c).
;; sqrt-shift: Num Num -> Num
;; Examples:
(check-expect (sqrt-shift 7 3) 2)
(check-expect (sqrt-shift 125 4) 11)

(define (sqrt-shift x c)
  (sqrt (- x c)))
```

What inputs are invalid?

Additional contract requirements

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) returns the square root of (x - c).
;; sqrt-shift: Num Num -> Num
;; Requires: x - c >= 0
;; Examples:
(check-expect (sqrt-shift 7 3) 2)
(check-expect (sqrt-shift 125 4) 11)

(define (sqrt-shift x c)
  (sqrt (- x c)))
```

Tests using `check-expect`

You have probably noticed that our tests look like

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

What is going on?

`check-expect` is a built-in function 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 returns.
- The second is the correct answer, calculated by hand.

If the answer the function actually returns 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.

Inexact tests

Some functions return *inexact* answers. Often this means it is an irrational number.
For example:

```
(sqrt 2) => #i1.4142135623730951
```

In this case, `(check-expect test-expression true-value)` will not work.
Instead use `(check-within test-expression true-value max-error)`
For example,

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

Ex. Write one exact and one inexact test for `sqrt-shift`.

More Details are in the Style Guide

More details on the Design Recipe are in the Style Guide, available on the course website.

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!

The string data type: `Str`

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"`.

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:

```
(string-append "now" "here") => "nowhere"
(string-append "he"   "llo "   "how"   " R"   "U?") => "hello how RU?"
```

...determine how many characters are in a string:

```
(string-length "length") => 6
```

...chop up a string:

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

Exercise

Use **string-append** and **substring** to complete the function chop-word:

```
;; (chop-word s): select some pieces of s.
;; chop-word: Str -> Str
;; 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")
```

Some Useful String Functions

We can also:

...check if strings are in alphabetic order:

```
(string<? "pearls" "swine") => #true  ; "pearls" before "swine".
(string<? "pearls" "pasta") => #false ; first chars equal so compare next.
(string>? "kneel" "zod") => #false    ; "kneel" before "zod"
(string=? "pearls" "gems") => #false
```

...convert between numbers and strings

```
(number->string 42) => "42"
(string->number "3.14") => 3.14
```

Ex: Read the documentation on some of these functions.

Exercise

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.
;; describe-string: Str -> Str
;; Examples:
(check-expect (describe-string "foo") "The string 'foo' has length 3")
(check-expect (describe-string "") "The string '' has length 0")
```

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`.

Thinking about Implementation

You should now have a clear purpose, contract, and examples 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 about Implementation

Thinking through the problem, it needs to be something like this:

```
(define (swap-parts s)
  (string-append
   ...  ; an expression that is the back half of s
   ...  ; an expression that is the front half of s
   ))
```

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".

Helper Functions

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 shoulds suggest purposes.

> **!** Never call a helper function "`helper`"! Use meaningful names!

Put helper functions above any functions they help.
See the Style Guide for further details.

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

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

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

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

Duplicated Code

Probably you have written something similar to this:

```
;; (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 identical code appears twice: `(quotient (string-length s) 2)`. This snippet find 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 a 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 `mid` to do this.

Ex. Write `mid`. Remember to follow the design recipe (purpose, contract, examples).

Ex. Update `back-part` and `front-part` to use `mid`.

Finally our task is complete. Once we have tested all our helpers, we can start testing our main function, `swap-parts`. Always test helpers thoroughly before testing the function they help!

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

Exercise

Think about what this problem needs:

- Define some useful constants.
- Write the purpose and contract.
- Write examples and tests.

A solution is on the next slide.

Sample Solution

```
(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...
```

Design Recipe Overview

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*.
In comparison, the CS115 Style Guide is quite short.

Write the **purpose**, **contract**, and some **examples** before the implementation!

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`.

Further Reading: *How to Design Programs*, Section 3.

Before we begin the next module, please

- Read the Wikipedia entry on *Higher-order functions*.