The design recipe

Readings:

- HtDP, section 2.5
- Survival and Style Guides

Topics:

- Programs as communication
- The design recipe
- Using the design recipe
- Tests
- Contracts

Programs as communication

Every program is an act of communication:

- Between you and the computer

- Between you and yourself in the future

- Between you and others

Human-only comments in Racket programs:
from a semicolon (**;**) to the end of the line.

Some goals for software design

Programs should be:

compatible, composable, correct, durable, efficient, extensible,
flexible, maintainable, portable, readable, reliable, reusable,
scalable, usable, and useful.

The design recipe

- Use it for every function you write in CS 135.

- A development process that leaves behind written explanation of the development

- Results in a trusted (tested) function which future readers (you or others) can understand

The five design recipe components

Purpose: Describes what the function is to compute.

Contract: Describes what type of arguments the function consumes and what type of value it produces.

Examples: Illustrating the typical use of the function.

Definition: The Racket definition of the function (header & body).

Tests: A representative set of function applications and their expected values. Examples also serve as Tests.

Order of execution

The order in which you carry out the steps of the design recipe is very important. Use the following order:

- Write a draft of the Purpose

- Write Examples (by hand, then code)

- Write Definition Header & Contract

- Finalize the purpose with parameter names

- Write Definition Body

- Write Tests

Using the design recipe

Purpose (first draft):

;; produce the sum of the squares of two numbers

Examples:

$$3^2 + 4^2 = 9 + 16 = 25$$

;; Example:

(check-expect (sum-of-squares 3 4) 25)

Using the design recipe (cont)

Header & Contract:

;; sum-of-squares: Num Num $\rightarrow$ Num

(define (sum-of-squares n1 n2) ...)

Purpose (final draft):

;; (sum-of-squares n1 n2) produces the sum of the squares
;;   of n1 and n2.

Using the design recipe (cont)

Write Function Body:

(define (sum-of-squares n1 n2)
   (+ (sqr n1) (sqr n2)))

Write Tests:

;; Tests:
(check-expect (sum-of-squares 0 0) 0)
(check-expect (sum-of-squares -2 7) 53)
(check-expect (sum-of-squares 0 2.5) 6.25)

Using the design recipe (final result)

```
;; (sum-of-squares n1 n2) produces the sum of the ...
;; sum-of-squares: Num Num → Num
;; Examples:
(check-expect (sum-of-squares 3 4) 25)

(define (sum-of-squares n1 n2)
  (+ (sqr n1) (sqr n2)))
;; Tests:
(check-expect (sum-of-squares 0 0) 0)
(check-expect (sum-of-squares 0 2.5) 6.25)
```

Tests

- Tests should be written later than the code body.

- Tests can then handle complexities encountered while writing
  the body.

- Tests don't need to be "big".

  In fact, they should be small and directed.

- The number of tests and examples needed is a matter of
  judgement.

- **Do not** figure out the expected answers to your tests by running
  your program! Always work them out **independently**.

The teaching languages offer convenient testing methods:

```
(check-expect (sum-of-squares 3 4) 25)
(check-within (sqrt 2) 1.414 .001)
(check-error (/ 1 0) "/: division by zero")
```

check-within should only be used for inexact values.

Tests written using these functions are saved and evaluated at the
very end of your program.

This means that examples can be written as code.

Contracts

- We will be more careful than HtDP and use abbreviations.

  - Num: any Racket numeric value

  - Int: restriction to integers

  - Nat: restriction to natural numbers (0, 1, 2, 3, ...)

  - Any: any Racket value

- We will see more types soon.

Use the most specific type available.

Additional contract requirements

If there are important constraints on the parameters that are not fully described in the contract, add an additional **requires** section to "extend" the contract.

```
;; (my-function a b c) ...
;; my-function: Num Num Num → Num
;; requires: 0 < a < b
;;           c must be non-zero
```

Racket does not enforce contracts, which are just comments, and ignored by the machine.

Each value created during the running of a program has a type (integer, Boolean, etc.).

Types are associated with values, not with constants or parameters.

```
(define p 5)
(define q (mystery-fn 5))
```

This is known as **dynamic typing**.

Many other mainstream languages use **static typing** in which constants, parameters and values all have specified types. Constants and parameters of one type may not hold a value of an incompatible type.

With static typing, the header of a function might look like this:

```
Int foo(c:Num, g:Nat)
```

Here the contract is part of the language.

A program containing the function application `foo(65, 100.0)` would be illegal.

Dynamic typing is a potential source of both flexibility (as we will see) and confusion.

Contracts are important in keeping us unconfused. However, they are only human-readable comments and are not enforced by the computer.

We can also create functions that check their arguments to catch type errors more gracefully (examples soon).

Unless stated otherwise, **you may assume that all arguments provided to a function will obey the contract** (including our automated testing).

Design recipe style guide

Note that in these slides, sections of the design recipe are often omitted or condensed because of space considerations.

Consult the course style guide before completing your assignments.

The style guide is used for multiple courses; only Pages 1–19 apply to CS135.

Goals of this module

You should understand the reasons for each of the components of the design recipe and the particular way that they are expressed.

You should start to use the design recipe and appropriate coding style for all Racket programs you write.