CS 135
September 6, 2019
The design recipe
Using the design recipe
Tests & contracts

The Design Recipe
Readings

• HtDP, section 2.5
• Survival and Style Guides
The design recipe
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.

\((sqr 3) \; \text{this lines produces 9}\)
Some goals for software design

Programs should be:

- compatible,
- composable,
- correct,
- durable,
- efficient,
- extensible,
- flexible,
- maintainable,
- portable,
- readable,
- reliable,
- reusable,
- scalable,
- usable, and
- useful.
Some goals for software design – Software does the right thing

Programs should be:

- compatible,
- composable,
- **correct**,
- durable,
- efficient,
- extensible,
- flexible,
- maintainable,
- portable,
- readable,
- **reliable**,
- reusable,
- scalable,
- usable, and
- **useful**.
Some goals for software design – Software follows good coding style

Programs should be:

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

The Design recipe is a development process that leaves behind written explanation of the development. Following its approach results in a trusted (tested) function that future readers (you or others) can use with confidence, understand, and extend if necessary.
The five components of the Design recipe

1. **Purpose:** Describes what the function is to compute.

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

3. **Examples:** Illustrating the typical use of the function.

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

5. **Tests:** A representative set of function applications and their expected values. Examples also serve as Tests.
Using the design recipe
Step-by-step execution for the Design Recipe

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

1. Write a draft of the **Purpose**
Application example – Purpose

Purpose (first draft):

;;; produces the sum of the squares of two numbers
Step-by-step execution for the Design Recipe

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

1. Write a draft of the **Purpose**
2. Write **Examples** (by hand, then code)
Application example – Examples

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

;; Example:
(check-expect (sum-of-squares 3 4) 25)
Step-by-step execution for the Design Recipe

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

1. Write a draft of the **Purpose**
2. Write **Examples** (by hand, then code)
3. Write **Contract & Definition Header**
Application example – Contract & Definition Header

Contract & definition header:

```scheme
;; sum-of-squares: Num Num -> Num
(define (sum-of-squares n1 n2)
  ...
)
```
Step-by-step execution for the Design Recipe

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

1. Write a draft of the **Purpose**
2. Write **Examples** (by hand, then code)
3. Write **Contract & Definition Header**
4. Finalize the **Purpose** with parameter names
Application example – Finalize Purpose

Purpose (first draft):

;;; produces the sum of the squares of two numbers

Purpose (final draft):

;;; (sum-of-squares n1 n2) produces the sum of the squares of n1 and n2.
Step-by-step execution for the Design Recipe

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

1. Write a draft of the **Purpose**
2. Write **Examples** (by hand, then code)
3. Write **Contract & Definition Header**
4. Finalize the **Purpose** with parameter names
5. Write **Definition Body**
Application example – Definition body

Definition header:

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

Definition body:

(define (sum-of-squares n1 n2)
  (+ (sqr n1) (sqr n2)))
Step-by-step execution for the Design Recipe

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

1. Write a draft of the **Purpose**
2. Write **Examples** (by hand, then code)
3. Write **Contract & Definition Header**
4. Finalize the **Purpose** with parameter names
5. Write **Definition Body**
6. Write **Tests**
Application example – Tests

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)
Step-by-step execution for the Design Recipe

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

1. Write a draft of the **Purpose**
2. Write **Examples** (by hand, then code)
3. Write **Contract & Definition Header**
4. Finalize the **Purpose** with parameter names
5. Write **Definition Body**
6. Write **Tests**

Use it for every function you write in CS 135!
Application example – Final result

;; (sum-of-squares n1 n2) produces the sum of the squares of n1 and n2.
;; 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 -2 7) 53)
(check-expect (sum-of-squares 0 2.5) 6.25)
Tests & Contracts
Tests

Tests should be written later than the code body.

Tests can then handle complexities encountered while writing the body.

Tests do not 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.
Tests

The teaching languages offer convenient testing methods:

(check-expect (sum-of-squares 3 4) 25) ; exactly 25
(check-within (sqrt 2) 1.41 0.01) ; 1.41 +/- 0.01
(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.
Contract

We will be more careful than HtDP and use abbreviations.

- **Any**: any Racket value
- **Num**: any Racket numeric value
- **Int**: restriction to integers (...,-2,-1,0,1,2,...)
- **Nat**: restriction to natural numbers (0,1,2,3,4,...)

Use the **most specific type available**.

We will see more types soon.
**Contract – Additional 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.

```plaintext
;; (my-function a b c) does weird stuff
;; 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 therefore ignored by Racket.
Contract – Types

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-func 5))

This is known as dynamic typing.
Contract – Types

Many other mainstream languages use a more restrictive approach known as *static typing*.

With static typing, the header of our `ok-grade` function might look like this:

```lisp
Bool ok-grade(Sym c, Nat g)
```

Here, the contract is part of the language.

A program containing the function application `ok-grade(65, 'CS136)` would be illegal.
Contract – Types

Static typing (e.g., C++):

\[ \text{Bool} \quad \text{ok-grade}(\text{Sym } c, \text{ Nat } g) \]

\text{ok-grade}(65, \ 'CS136') \text{ is illegal (it would result in a compile-time error).}

Dynamic typing (here: Racket):

\[ \text{ok-grade: Sym Nat-> Bool} \]

\text{(define (ok-grade \ c \ g)}

\text{(ok-grade 65 \ 'CS136) \text{ is legal (it could, however, result in a runtime error).}}
Contract – Types

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.
The Design Recipe – End of module
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.