Module 03: 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.
Some goals for software design

We emphasize the following...

- **Correctness**: Does it give the right output? Does it meet the specification? ⇒ starting now.
- **Efficiency**: Does it minimize the use of computer resources such as processor time or memory usage? ⇒ starting 2nd year
- **Readability**: Can another programmer easily understand it?
- **Reliability**: Does it crash? Does it always work?
- **Flexibility**: Is it easy to change?
- **Extensibility**: Is it easy to add new features?
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 design recipe

• For simple programs it almost seems like you do not need to go through all this effort.

• At some point: maybe in this course, maybe in 2nd year, you’ll reach a point where you start making mistakes because the complexity of the program is too large.

• Develop good habits now.
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.
The design recipe

• The **Purpose** is more of a *description in English* whereas the **Contract** which is more of a *mathematical description*.

• **Consume** describes the parameters.

• **Produce** is what it returns, i.e. the result of applying the function.

• The **header** is the function name, parameters, and the **define** keyword.

• The **body** is the expression that computes the result.
The design recipe

• If you want help from course staff for handling problems with your code, you need to provide them with the functions contract, purpose, and examples.

• We will be using a new function: check-expect checks to see if a function has returned the expected results.

• Generally expressions can only use functions defined before the expression. This is not the case for check-expect.
Order of execution

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 Definition Header & Contract
4. Finalize the purpose with parameter names
5. Write Definition Body
6. 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 → 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 *three 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, ...)
    In this course natural numbers include 0.
  – **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))

In the code above, we have no idea what type is associated with the value of q without tracing through the evaluation of (mystery-fn 5).
Racket’s approach is known as **dynamic typing**, i.e. types are checked as the code is executed.

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:

```plaintext
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 and confusion*, as we will see.

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 marking scheme is typically

- 5% for readability (format, meaningful variable names, etc.)
- 5% for purpose and contract
- 10% for tests and examples, combined
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.
Module 03 Summary

The Design Recipe

2. The **header** is the function name, parameters, and the `define` keyword. [5.1]
3. The **body** is the expression that computes the result. [5.1]
4. Three new functions for testing were introduced: `check-expect`, `check-within` and `check-error`. [12]
5. Contracts may specify the following types: `Num`, `Int`, `Nat`, `Any` with more types to come later. [13]
Module 03 Summary

Requires and Types

6. Use requires to specify any additional constraints on the parameters not covered in the contract. [14]

7. **Dynamic typing** is when types are checked as the code is executed. [16]

8. **Static typing** is when constants, parameters and values and the values that functions produce all have specified types. [16]