Module 2: The design recipe

Readings:

- HtDP, Sections 1-3
- Survival Guide on assignment style and submission

We are covering the same material as the text, but using different examples. Lectures do not cover all the details, so keeping up with the readings is important.

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

Comments

Comment: information important to humans but ignored by the computer

Code: anything that is not a comment; machine-readable

```
;; Constants for calendars
(define year-days 365) ; not a leap year
```

Use one semicolon to indicate that the rest of the line is a comment.

Use two semicolons to start a line that is a comment.

Do not use DrRacket's Comment Boxes.

Goals for software design

Misconception: correctness and speed only.

Partial list of goals for programs: compatible, composable, *correct*, durable, *efficient*, *extensible*, *flexible*, maintainable, portable, *readable*, *reliable*, reusable, *scalable*, usable, and useful.

The design recipe

A development process that leaves behind a written explanation of the development.

Results:

- reliability (the function has been tested)
- readability (comments make it understandable)

You will use the design recipe for every function throughout the course.

The basic recipe will be modified as we add new ingredients.

The three comment components

Contract: a formal description of what type of arguments the function consumes and what type of value it produces.

;; fun-name: type1 type2 ... typek $\rightarrow$ type-produced

Purpose: a description of what the program is to produce; the meaning of each parameter to the function should be made evident.

Examples: illustrations of the use of the program, chosen to aid in the development of the function.

;; Examples: (fun-name argument) $\Rightarrow$ produced

The two code components

Definition: the Scheme function definition, consisting of the **function header** (the first line, with the names of the function and the parameters) and the **body**.

Tests: a set of inputs and expected outputs, each designed to test a specific aspect of the function.

Components should appear in this order: contract, purpose, examples, definition, tests.

Components are written in a slightly different order.

Using the design recipe

Step 1: **Contract** and **function header**. The number and order of parameters should match.

Step 2: **Purpose**. Each parameter should be mentioned by name.

Step 3: **Examples**. They should capture all possible cases.

Step 4: **Body**.

Step 5: **Tests**.

Step 6: Run the program.

Never cut and paste from Interactions to Definitions.

Types in contracts

Our rules will be more strict than those used in the textbook.

- Scheme data types: num (any numeric value), image.
- User-defined data types.
- int for any integer, such as in quotient.
- nat for any natural number (including 0) (used later).
- any for any Scheme value.
- We will add to this list; see Style Guide for details.

Design recipe example

Goal: a function that squares two numbers and sums the results.

Mathematically: sum-of-squares: $\mathbb{R} \times \mathbb{R} \rightarrow \mathbb{R}$

Step 1: **Contract** and **function header**

```
;; sum-of-squares: num num →  num
(define (sum-of-squares arg1 arg2)
```

Step 2: **Purpose**

```
;; Purpose: produces sum of squares of arg1 and arg2.
```

Step 3: **Examples**

```
;; Examples: (sum-of-squares 3 4) ⇒ 25
;; (sum-of-squares 0 2.5) ⇒ 6.25
```

Step 4: **Body** (added to function header)

```
(define (sum-of-squares arg1 arg2)
    (+ (* arg1 arg1) (* arg2 arg2)))
```

```
;; sum-of-squares: num num → num
;; Purpose: produces sum of squares of arg1 and arg2.
;; Examples: (sum-of-squares 3 4) ⇒ 25
;; (sum-of-squares 0 2.5) ⇒ 6.25
(define (sum-of-squares arg1 arg2)
    (+ (* arg1 arg1) (* arg2 arg2)))
```

In your work:

Single arrow as ‘minus’ and ‘greater-than’ (−>)

Double arrow as ‘equals’ and ‘greater-than’ (=>)

Testing in DrRacket

Use (check-expect fun-application value-expected).

Each test is targeted to focus on a specific aspect of the code.

Tests typically include all the examples, plus some more.

Tests don't need to be "big". Smaller is often better, because it is easier to figure out what went wrong if they fail.

The number needed is a matter of judgement.

Figure out expected answers by hand, not by running the program.

```
;; sum-of-squares: num num → num
;; Purpose: produces sum of squares of arg1 and arg2.
;; Examples: (sum-of-squares 3 4) ⇒ 25
;; (sum-of-squares 0 2.5) ⇒ 6.25
(define (sum-of-squares arg1 arg2)
    (+ (* arg1 arg1) (* arg2 arg2)))
;; Tests for sum-of-squares:
(check-expect (sum-of-squares 3 4) 25)
(check-expect (sum-of-squares 0 2.5) 6.25)
(check-expect (sum-of-squares -1 1) 2)
```

Note: check-expect only works with exact numbers.

The following code will result in an error.

```
(define (circle-area r) (* pi r r))
(check-expect (circle-area 1) pi)
```

Use check-within with a function application, a value, and a range within which the answer can differ from the value provided.

A range of .00001 should suffice.

```
(check-within (circle-area 1) pi .00001)
```

The data type string

A **string** 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"

See the course Web site (not in the textbook).

Exchanging front and back of a string

swap-parts will exchange the front and back of a string.

The *front* is the first half of the string.

The *back* is the second half of the string.

If the length of the string is odd, we'll make the front shorter.

```
;; swap-parts: string → string
(define (swap-parts str)
```

```
;; swap-parts: string → string
;; Purpose: produces a string formed by swapping the
;; front and back of str.
;; Examples: (swap-parts "angle") ⇒ "glean"
;; (swap-parts "potshots") ⇒ "hotspots"
(define (swap-parts str)
```

Subtasks: finding the midpoint, extracting the front, extracting the back.

Helper functions

A **helper function** is a function that is used in the main function to

- generalize similar expressions,
- express repeated computations, and
- avoid long, unreadable function definitions.

Choose meaningful function names for all functions and parameters, including helper functions (not helper).

Use the full design recipe to create helper functions.

Put helper functions before the main function.

Indentation and style guidelines available on the course Web site.

Finding the midpoint

```
;; mid: string → nat
;; Purpose: produces the integer part of (string-length str)/2.
;; Examples: (mid "cs115") ⇒ 2
;; (mid "Hi") ⇒ 1
(define (mid str)
  (quotient (string-length str) 2))
;; Tests for mid
(check-expect (mid "cs115") 2)
(check-expect (mid "Hi") 1)
```

Extracting the front

```
;; front-part: string → string
;; Produces the front part of str.
;; Examples: (front-part "angle") ⇒ "an"
;; (front-part "potshots") ⇒ "pots"
(define (front-part str)
  (substring str 0 (mid str)))
;; Tests for front-part
(check-expect (front-part "angle") "an")
(check-expect (front-part "potshots") "pots")
```

Extracting the back

```
;; back-part: string → string
;; Produces the back part of str.
;; Examples: (back-part ”angle”) ⇒ ”gle”
;; (back-part ”potshots”) ⇒ ”hots”
(define (back-part str)
  (substring str (mid str)
             (string-length str)))
;; Tests for back-part
(check-expect (back-part "angle") "gle")
(check-expect (back-part "potshots") "hots")
```

Developing swap-parts

```
;; swap-parts: string → string
;; Purpose: produces a string formed by swapping the back
;; and front of str.
;; Examples: (swap-parts "angle") ⇒ "glean"
;; (swap-parts "potshots") ⇒ "hotspots"
(define (swap-parts str)
  (string-append (back-part str) (front-part str)))
;; Tests for swap-parts
(check-expect (swap-parts "angle") "glean")
(check-expect (swap-parts "potshots") "hotspots")
```

Computing a phone bill

cell-bill consumes the numbers of daytime and evening minutes used and produces the total charge.

Details of the phone plan:

- 100 free daytime minutes
- 200 free evening minutes
- 1 dollar per minute charge for each additional daytime minute
- 50 cent per minute charge for each additional evening minute

How can we remember the meaning of each number?

Using constants

```
;; Constants for phone plan
;; Free limits
(define day-free 100)
(define eve-free 200)
;; Rates per minute
(define day-rate 1)
(define eve-rate .5)
```

Use constants for readability and flexibility.

Put them before helper functions and main functions.

```
;; cell-bill: nat nat → num
;; Purpose: produces cell phone bill for day and eve minutes.
;; Examples: (cell-bill 101 0) ⇒ 1
;; (cell-bill 99 0) ⇒ 0
;; (cell-bill 0 199) ⇒ 0
;; (cell-bill 0 202) ⇒ 1
;; (cell-bill 150 300) ⇒ 100
(define (cell-bill day eve)
```

Developing charges-for

```
;; charges-for: nat nat num → num
;; Purpose: produces charges for minutes,
;; given the rate per minute past the freelimit.
;; Examples: (charges-for 101 100 5) ⇒ 5
;; (charges-for 99 100 34) ⇒ 0
(define (charges-for minutes freelimit rate)
  (max 0 (* (- minutes freelimit) rate)))
;; Tests for charges-for:
(check-expect (charges-for 101 100 5) 5)
(check-expect (charges-for 99 100 34) 0)
```

Completing cell-bill

```
(define (cell-bill day eve)
  (+
   (charges-for day day-free day-rate)
   (charges-for eve eve-free eve-rate)))
;; Tests for cell-bill
(check-expect (cell-bill 101 0) 1)
(check-expect (cell-bill 99 0) 0)
(check-expect (cell-bill 0 199) 0)
(check-expect (cell-bill 0 202) 1)
(check-expect (cell-bill 150 300) 100)
```

Design recipe summary

1. **Contract** and **function header**. The number and order of parameters should match.
2. **Purpose**. Each parameter should be mentioned by name.
3. **Examples**. They should capture all possible cases.
4. **Body**. Identify constants and helper functions. Use design recipe on helper functions.
5. **Tests**. Choose check-expect or check-within. Figure out results by hand.
6. Run the program.

Goals of this module

You should be comfortable with these terms: comment, code, contract, purpose, examples, definition, function header, body, tests, helper function.

You should know the types that are allowed in contracts so far.

You should understand the reasons for each of the components of the design recipe, the order in which they appear, and the order in which they should be created.

You should know when to use check-expect and when to use check-within.

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

You should look for opportunities to use helper functions and constants to structure your programs, and gradually learn when and where they are appropriate.

You should be able to use strings in labs, assignments, and exams.