The Design Recipe
Fall 2019

Contents

1 Design Recipe Steps 1

2 A ReasonML Example 6

1 Design Recipe Steps

This PDF outlines the steps to writing the design recipe; each step is accompanied by part of an example for writing the design recipe. The example function is in Racket, and it determines whether or not a database is empty.

1. Provide data definitions for all non-atomic data types. (If multiple procedures in a single file use the same non-atomic types, provide these data definitions once at the top of the file, not once-per-procedure).

```racket
;; Data Definition
;; Example data:
;; string: "Carmen", "Audrey", "dog"
;;
;; A database is a list of strings
```

2. Provide examples of the data the procedure will process and produce.

```racket
;; Data Definition
;; Example data:
;; string: "Carmen", "Audrey", "dog"
;;
;; A database is a list of strings
;;
;; Examples:
;; (define db1 empty)
;; (define db2 (cons "cat" (cons "dog" empty)))

;; bool: true, false
```
3. Specify the procedure’s **type signature**, which describes the type of data the procedure consumes, and the type it produces. Your types should be meaningful Racket types, like ‘string’, ‘int’, ‘real’, and ‘<type>list’. A more specific type with no semantic meaning in Racket—something like ”nonnegative integer”—should be specified in the I/O contract rather than the type signature. A procedure consuming nonnegative integers would just have the input type ”int” and the more detailed description of the nonnegative constraint in the I/O contract.

```racket
;;; Data Definition
;;; Example data:
;;; string: "Carmen", "Audrey", "dog"
;;; A database is a list of strings
;;; Examples:
(define db1 empty)
(define db2 (cons "cat" (cons "dog" empty)))

;;; bool: true, false
;;; is-database-empty?: database -> bool
```

4. Following the type signature, describe the procedure’s **call structure**, i.e., give names to the procedure and its arguments. (This involves writing the start of a Racket program rather than writing a comment.)

```racket
;;; Data Definition
;;; Example data:
;;; string: "Carmen", "Audrey", "dog"
;;; A database is a list of strings
;;; Examples:
(define db1 empty)
(define db2 (cons "cat" (cons "dog" empty)))

;;; bool: true, false
;;; is-database-empty?: database -> bool

(define (is-database-empty? db) ...
```
5. Write a **specification** for the procedure. That is, in words, not code, state the relationship between the procedure's input and output (make sure to use the argument names you created in the call structure). This goes in a comment above the call structure. Writing the spec may include restricting the domain in the description of some variable, for instance declaring an input as nonnegative.

```scheme
;; Data Definition
;; Example data:
;; string: "Carmen", "Audrey", "dog"
;;
;; A database is a list of strings
;;
;; Examples:
(define db1 empty)
(define db2 (cons "cat" (cons "dog" empty)))

;; bool: true, false
;;
;; is-database-empty?: database -> bool
;;
;; input: db, a database
;; output: a boolean that is true if db is empty, and false if it is not
(define (is-database-empty? db)
  ...
)
```

6. Provide **test cases** that exemplify the procedure's operation. These tests must follow its call structure and satisfy its specification.

```scheme
;; Data Definition
;; Example data:
;; string: "Carmen", "Audrey", "dog"
;;
;; A database is a list of strings
;;
;; Examples:
(define db1 empty)
(define db2 (cons "cat" (cons "dog" empty)))

;; bool: true, false
;;
;; is-database-empty?: database -> bool
;;
;; input: db, a database
;; output: a boolean that is true if db is empty, and false if it is not
(define (is-database-empty? db)
  ...
)

;; Test cases
(check-expect (is-database-empty? empty) true)
(check-expect (is-database-empty? (cons "hedwig" empty)) false)
```
7. Write the template for the procedure based on the data definition and the type signature. This is where you would decide whether to use an if-statement or a cond-expression.

```scheme
;; Data Definition
;; Example data:
;; string: "Carmen", "Audrey", "dog"
;;
;; A database is a list of strings
;;
;; Examples:
;; (define db1 empty)
;; (define db2 (cons "cat" (cons "dog" empty)))

;; bool: true, false
;;
;; is-database-empty?: database -> bool
;;
;; input: db, a database
;; output: a boolean that is true if db is empty, and false if it is not

(define (is-database-empty? db)
  (cond
    [(empty? db) ...]
    [(cons? db) ...]))

;; Test cases
(check-expect (is-database-empty? empty) true)
(check-expect (is-database-empty? (cons "hedwig" empty)) false)
```
8. **Code** the procedure. Specifically, fill in the template clause by clause. For each of the possible input types, decide which fields in the input structures are relevant to the problem at hand, and figure out how to operate on them to generate the desired output.

```scheme
;; Data Definition
;; Example data:
;; string: "Carmen", "Audrey", "dog"
;;
;; A database is a list of strings
;;
;; Examples:
;; (define db1 empty)
;; (define db2 (cons "cat" (cons "dog" empty)))

;; bool: true, false
;;
;; is-database-empty?: database -> bool
;;
;; input: db, a database
;; output: a boolean that is true if db is empty, and false if it is not

(define (is-database-empty? db)
  (cond
   [(empty? db) true]
   [(cons? db) false]))

;; Test cases
(check-expect (is-database-empty? empty) true)
(check-expect (is-database-empty? (cons "hedwig" empty)) false)
```

9. **Run** your program on your test cases.
2 A ReasonML Example

```reasonml
type database = list(string);

/* Data Definition */
/* Example data: */
/* A database is a list of strings */
/* Examples: */

/* */
/* [] ; */
/* ["cat", "dog"] ; */

/* input: db, a database */
/* output: a boolean - true if db is empty, and false if it is not */

/* note that if this was a recursive function, */
/* we would need a recursion diagram */

let isDatabaseEmpty: database => bool = db =>
    switch(db) {
        |[] => true
        |_  => false
    };

/* test cases */
check_expect (isDatabaseEmpty ([]), true);
check_expect (isDatabaseEmpty ("the", "design", "recipe")), false);
```

Please let us know if you find any mistakes, inconsistencies, or confusing language in this or any other CS 17 document by filling out the anonymous feedback form: [http://cs.brown.edu/courses/csci0170/feedback](http://cs.brown.edu/courses/csci0170/feedback).