HtD Functions

Help

The How to Design Functions (HtDF) recipe is a **design method** that enables systematic design of functions. We will use this recipe throughout the term, although we will enhance it as we go to solves more complex problems.

The HtDF recipe consists of the following steps:

1. Signature, purpose and stub.
2. Define examples, wrap each in `check-expect`.
3. Template and inventory.
4. Code the function body.
5. Test and debug until correct

NOTE:

- Each of these steps build on the ones that precede it. The signature helps write the purpose, the stub, and the check-expects; it also helps code the body. The purpose helps write the check-expects and code the body. The stub helps to write the check-expects. The check-expects help to code the body as well as to test the complete design.
- It is sometimes helpful to do the steps in a different order. Sometimes it is easier to write examples first, then do signature and purpose. Often at some point during the design you may discover an issue or boundary condition you did not anticipate, at that point go back and update the purpose and examples accordingly. But you should never write the function definition first and then go back and do the other recipe elements -- for some of you that will work for simple functions, but you will not be able to do that for the complex functions later in the course!
- Throughout the HtDF process be sure to "run early and run often". Run your program whenever it is well-formed. The more often you press run the sooner you can find mistakes. Finding mistakes one at a time is much easier than waiting until later when the mistakes can compound and be more confusing. Run, run, run!

Signature, purpose and stub.

Write the function signature, a one-line purpose statement and a function stub.

A signature has the type of each argument, separated by spaces, followed by `->`, followed by the type of result. So a function that consumes an image and produces a number would have the signature `Image -> Number`.

Note that the stub is a syntactically complete function definition that produces a value of the right type. If the type is Number it is common to use 0, if the type is String it is common to use "a" and so on. The value will not, in general, match the purpose statement. In the example below the stub produces 0, which is a Number, but only matches the purpose when double happens to be called with 0.

```
;; Number -> Number
;; produces n times 2

(define (double n)  0)  ; this is the stub
```

The purpose of the stub is to serve as a kind of scaffolding to make it possible to run the examples even before the function design is complete. With the stub in place check-expects that call the function can run. Most of them will fail of course, but the fact that they can run at all allows you to ensure that they are at least well-formed: parentheses are balanced, function calls have the proper number of arguments, function and constant names are correct and so on. This is very important, the sooner you find a mistake -- even a simple one -- the easier it is to fix.

Define examples, wrap each one in check-expect.

Write at least one example of a call to the function and the expected result the call should produce.

You will often need more examples, to help you better understand the function or to properly test the function. (If once your function works and you run the program some of the code is highlighted in black it means you definitely do not have enough examples.) If you are unsure how to start writing examples use the combination of the function signature and the data definition(s) to help you generate examples. Often the example data from the data definition is useful, but it does not necessarily cover all the important cases for a particular function.

The first role of an example is to help you understand what the function is supposed to do. If there are boundary conditions be sure to include an example of them. If there are different behaviours the function should have, include an example of each. Since they are examples first, you could write them in this form:

```
;; (double 0) should produce 0
;; (double 1) should produce 2
;; (double 2) should produce 4
```

When you write examples it is sometimes helpful to write not just the expected result, but also how it is computed. For example, you might write the following instead of the above:

```
;; (double 0) should produce (* 0 2)
;; (double 1) should produce (* 1 2)
;; (double 2) should produce (* 2 2)
```

While the above form satisfies our need for examples, DrRacket gives us a better way to write them, by enclosing them in check-expect. This will allow DrRacket to check them automatically when the function is complete. (In technical terms it will turn the examples into unit tests.)

```
;; Number -> Number
;; produces n times 2
(check-expect (double 0) (* 0 2))
(check-expect (double 1) (* 1 2))
(check-expect (double 3) (* 3 2))

(define (double n)  0)  ; this is the stub
```

The existence of the stub will allow you to run the tests. Most (or even all) of the tests will fail since the stub is returning the same value every time. But you will at least be able to check that the parentheses are balanced, that you have not misspelled function names etc.

Template and inventory

Before coding the function body it is helpful to have a clear sense of what the function has to work with -- what is the contents of your bag of parts for coding this function? The template provides this.

Starting in week 2 templates are produced as part of the How to Design Data Definitions (HtDD) recipe, by following the rules on the Data Driven Templates web page. You should copy the template from the data definition to the function design, rename the template, and write a comment that says where the template was copied from. Note that the template is copied from the data definition for the consumed type, not the produced type.

For primitive data like numbers, strings and images the body of the template is simply `(... x)` where x is the name of the parameter to the function.

Once the template is done the stub should be commented out.

```
;; Number -> Number
;; produces n times 2
(check-expect (double 0) (* 0 2))
(check-expect (double 1) (* 1 2))
(check-expect (double 3) (* 3 2))

;(define (double n) 0) ; this is the stub

(define (double n)     ; this is the template
  (... n))
```

It is also often useful to add constant values which are extremely likely to be useful to the template body at this point. For example, the template for a function that renders the state of a world program might have an MTS constant added to its body. This causes the template to include an inventory of useful constants.

Code the function body

Now complete the function body by filling in the template.

Note that:

- the signature tells you the type of the parameter(s) and the type of the data the function body must produce
- the purpose describes what the function body must produce in English
- the examples provide several concrete examples of what the function body must produce
- the template tells you the raw material you have to work with

You should use all of the above to help you code the function body. In some cases further rewriting of examples might make it more clear how you computed certain values, and that may make it easier to code the function.

```
;; Number -> Number
;; produces n times 2
(check-expect (double 0) (* 0 2))
(check-expect (double 1) (* 1 2))
(check-expect (double 3) (* 3 2))

;(define (double n) 0) ; this is the stub
```

```
;(define (double n)    ; this is the template
;  (... n))

(define (double n)
  (* n 2))
```

Test and debug until correct

Run the program and make sure all the tests pass, if not debug until they do. Many of the problems you might have had will already have been fixed because of following the "run early, run often" approach. But if not, debug until everything works.

Created Wed 20 Feb 2013 10:27 PM PST

Last Modified Tue 28 Jan 2014 2:43 PM PST